OpenID Connect Client Registration API for Federated Cloud Platforms

Erik Berdonces Bonelo

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Berlin 28.12.2016

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisors:

Prof. Dr. Axel Küpper

M.Sc. Mathias Slawik

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Erik Berdonces Bonelo

Title: OpenID Connect Client Registration API for Federated Cloud Platforms

Date: 28.12.2016 Language: English Number of pages: 5+63

Department of Computer Science

Professorship: Distributed Systems and Services

Supervisor: Assoc. Prof. Keijo Heljanko

Advisors: Prof. Dr. Axel Küpper, M.Sc. Mathias Slawik

Nowadays, information technology is a key driver in our world. Big cloud federations are aiming to increase their computing power and achieve better results while being scalable. This huge IT systems are managed by multiple users having different roles and at the same time, new services deployment automation is needed to be able to cope with the rising need of resources.
This flexibility in deployment has created concerns on the security and the maintainability of these extensive systems. These requisites have led to start CYCLONE platform, a project focused to provide authentication and authorization services towards services running under control of federated unions of users. CYCLONE, at the moment working as a proof of concept, now allows to authenticate and authorize access to users using one-click-deployment applications against their federation's credentials. However, actual SSO systems require registration of the services against their Identity Providers in order to provide user validation. In this master thesis, we present two the components of CYCLONE.
The first one is a service registration for clients of the OpenID Connect Single Sign-On protocol that allows newly deployed services to be registered automatically against CYCLONE's SSO component, using RedHat's Keycloak authentication solution. Based on the real world scenarios that defined the CYCLONE platform, we have designed and implemented a solution alternative to the ones provided by Keycloak, and to evaluate it we have compared it to Keycloak's alternatives. As a result we have created a simple API implementation from where it's possible to track who is executing this registrations of new clients, in comparison to the anonymous ones provided by other solutions.
The second one is a module that allows easy SSH authorization through the use of CYCLONE's SSO backend as identity provider and that has been evaluated and tested by one of CYCLONE's use cases.

Keywords: CYCLONE, federation, OpenID Connect, Keycloak, PAM, SSH

Contents

Abbreviations

SSO	Single Sign-On
IdP	Identity Provider
SP	Service Provider
UA	User Agent
JWT	JSON Web Token
OIDC	OpenID Connect
SAML	Security Assertion Markup Language
JPA	Java Persistence API
SPI	Service Provider Interfaces
VPP	Virtual Power Plant
DER	Distributed Energy Resource
UI	User Interface
DAO	Data Access Object
API	Application Program Interface
CLI	Command Line Interface
UC	Use Case

1 Introduction

This thesis presents and implements a solution to authenticate dynamically deployed services in a federated network of users into an SSO provider. This authentication solution is divided into two components: one server implementation which allows to register the clients automatically in the SSO provider and a client that provides authentication in the distributed services of the federation. Both of them together provide a solution of authentication and authorization in cloud systems based on distributed federations. This topic is studied through the different use cases provided by the CYCLONE platform which this work is built on.

In this introductory section, first the background and motivation of the study are presented. Second, the research objectives are defined followed with an overview on the structure of this thesis.

1.1 Background and Motivation

Nowadays, information technology is a key driver in our world. This explosion in web development has been integrated for example into the Internet of the Things (IoT). This trend of connecting everything physical has not only affected our homes but is also starting to affect private systems and existing structures. The development of these new technologies is impacting on how new systems are being developed and how old ones are getting obsolete. Thus, older traditional systems need to be updated to adapt to new paradigms that exist in the present and didn't exist in the past and that actual technologies can solve.

This introduces us to new challenges when creating our systems to manage complex infrastructures. These new paradigms include among others the location agnostic cooperation between different entities, where different users can cooperate in a common objective regardless to their location. However, this online interconnection has also increased the amount of attacks to steal information from secure services. Just as an example, it has been revealed that Yahoo got stolen 1 billion accounts in 2013 and Snowden uncovered NSA's tracking and gathering of personal data. These are not the only cases in security that have been relevant to people, but all together, people's awareness on data security has risen and governments have started to regulate data privacy through laws. For example, in the European Union, a new data protection regulation has been approved since April 2016 (Regulation (EU) 2016/679). This becomes critical in highly distributed systems where security is essential and that a misuse of the secured resources can enable grave legal implications and where multiple users can access to these resources.

CYCLONE platform, as one of Europe's Horizon 2020 projects aims to develop new systems that will cover our future necessities in terms of security. CYCLONE's aim is to provide secure one click deployments of applications and services. These deployments, usually controlled by an orchestration platform, need to process critical data such as sequenced genomic data or control or manage high available systems such as distributed power plants. At the same time, they need to be created easily to provide scalability for the system. Also, the access to this secure resources needs to

be handled by a community of users organized as a federation, which increases the chances that information can be leaked or hacked. As a result, there is a need to find a fair trade-off between security and deployment simplicity where user interaction is minimally needed.

This dynamic deployment of resources and services needs to be integrated with CYCLONE's Single Sign-On (SSO) service which provides authentication and authorization protocols. SSO technologies are a must in CYCLONE platform, as it helps to centralize all the user credential and permission management in a single service, essential when working with distributed services so we can avoid having to update all the individual services when any user's credentials get updated. CYCLONE's SSO technology is based on RedHat's Keycloak open source identity and access management solution. Keycloak implements OpenID Connect 1.0 and SAML 2.0 SSO technologies, actual industry standards in SSO technologies. SSO technologies require to register all the services using them to control the usage of user credentials and access to resources.

1.2 Research Objectives

As this solution is compounded by a server and its client, in this thesis we want to provide a solution for both cases. Thus, we want to propose two implementations that integrate together to provide a solution to the stated problem.

First, we want to analyze, propose and implement an OpenID Connect Client registration API for CYCLONE, a federated cloud platform. With it, we want to provide a system to register services against CYCLONE's SSO provider, so dynamically deployed services can use secure centralized authentication. Not only we need to create a system to register the clients automatically, but we also need to provide a service that may allow users to use this SSO technologies in their environments.

Second, we want to implement an integration of this SSO authentication into the actual existing environments provided by CYCLONE's use cases. Actual existing environments depend on SSH and X11 technologies to provide user interfaces. With our implementation, we aim to make this usually locally designed technologies capable of authenticating federated users in a distributed architecture such as CYCLONE.

The main aim of this thesis is to create a set of tools that can be deployed dynamically and can secure authenticate any federation's user into secure servers.

As CYCLONE is a project already in development, we need to study its structure. CYCLONE depends on eduGAIN federation to gather user credentials from different sources and proxies them towards the services requesting authentication. CYCLONE, in turn is also based in a set of use cases, providing real life production scenarios where the project should solve their given challenges. This use cases provided a set of user requirements, that define the direction towards our registration solution needs to evolve. However, CYCLONE's use cases are not the only ones relevant when implementing a solution in a federated system. Thus we also analyzed the requirements of other stakeholders so our platform can be deployed in locations under regulation. These stakeholders include Technische Universität Berlin, SURFnet

national research and education network of the Netherlands and eduGAIN, as we are using their credentials in our platform. The analysis of the already created structure of CYCLONE will be the base of our implementation.

Also, we need to understand the technologies being used in CYCLONE and what are SSO strategies for user verification. For this, we need to analyze the structure of the platform and how it interacts with the eduGAIN federation to authenticate users in the deployed services. Furthermore, we need to list the features of Keycloak, and the different automation and configuration opportunities that the solution provides. As Keycloak includes two different SSO technologies, OpenID Connect (OIDC) and Security Assertion Markup Language 2.0 (SAML2.0), we need also to compare them and define their usage in our implementation. Bot technologies are similar because both provide authentication and authorization strategies but at the same time integrate different features and implementations. Thus, we must describe the different workflows they provide to verify users and compare the characteristics of each technology to allow access to the user's sensitive data. After an exhaustive background research, we can take a proper decision on how to create a dynamic registration system. To evaluate our results in the registration API, we compare our implementation with the one provided by Keycloak, as by now, at the end of the implementation, Keycloak has updated their registration API with new features. We contrast and discuss the security aspects that both provide and the advantages and disadvantages of each solution.

In parallel, we study and implement the different possibilities to allow a SSH protocol login to use CYCLONE's SSO service as authentication backend. To do so, we analyze the options we have to customize SSH, such as Keyboard-Interactive sessions and PAM modules, and how we can integrate the API calls and authentication against the OpenID Connect endpoint. The main objective of this implementation is to avoid using SSH RSA keys in favour of SSO authentication. Then, we use the feedback provided by IFB, one of CYCLONE's use cases, to evaluate our implementation in comparison to their requirements.

Finally, we provide an overview on how both component integrate together plus insights on the future implementations that we can do to improve this project.

1.3 Structure of the Thesis

This thesis is structured as follows. Chapter 2 presents a background review on the structure of CYCLONE project, which this thesis is based on and that is used to frame the study. In Chapter 3, we provide a study of the single sign on technologies used to implement our system that is then used in Chapters 4 and 5 to design the implementation. The process of development and implementation of the registration client API is explained in Chapter 4 and the client to use the SSO login from already existing environments is defined in Chapter 5. Lastly, the evaluations and conclusions of the study are presented in Chapters 6 and 7.

2 Application environment

This chapter describes the environment and use cases in which this thesis has been based on. First we explain what is the CYCLONE platform and which is its current status. Then, we outline the different use cases that inspired this project. Finally, we illustrate the requirements required for this thesis because of related parties needs.

2.1 CYCLONE platform

CYCLONE platform [1][2] is a project part of the Horizon 2020 Programme of the European Union. Its objective is to create a simple platform that will allow scientists from different research institutions to share resources in a secure way. Authentication in the system is done through the validation of the scientists' credentials provided by the research institutions. These credentials are obtained through the eduGAIN federation [3] and proxied through CYCLONE towards the internal cloud services of the research institution.

The CYCLONE platform depends on a set of previously developed components which interact together: SlipStream, StratusLab, TCTP, OpenNAAS. These components help to deploy and orchestrate the web and authentication services that make CYCLONE work in each individual private cloud.

2.1.1 eduGAIN Federation

CYCLONE depends on the eduGAIN federation to provide a list of education institutions and federations Identity Providers. eduGAIN [4] stands for "EDUcation Global Authentication INfrastructure" and consists of a federation of federations of education and research institutions. eduGAIN enables an updated list of Identity Providers metadata of all the members of the federation and allows a trustworthy exchange of identity, authentication and authorization of the federation's users to any of the registered Service Providers. It consists on more than 1,500 Identity Providers, 1,000 Service Providers found through over 40 different federations [5].

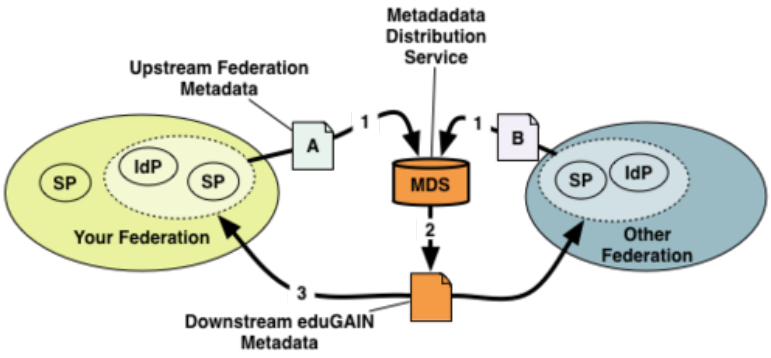

Figure 1: Diagram representing the metadata workflow in eduGAIN federation (from eduGAIN's wiki https://wiki.edugain.org/Metadata_Flow_in_eduGAIN)

eduGAIN uses the SAML 2.0 protocol to grant the authentication and authorization to the different Identity Providers of the federation. eduGAIN requires some procedure to join the federation, including fulfilling declarations of privacy, providing a Federation Policy and selecting two members as delegates inside the federation. Also, each of the federations which are part of eduGAIN need to provide an updated metadata configuration describing how to connect to their SAML 2.0 endpoints, and need to conform some tests to determine if the federation's configuration is valid.

Just for a single node in the eduGAIN federation, this can be a feasible set of requirements [5]. However, in case any institution wants to provide access to the federation's data to any of its services, it needs to create a new SAML 2.0 endpoint and accept the previously fulfilled data. In case of simple services, this means a lot of time and bureaucracy in order to setup a new service inside the federation.

This is why CYCLONE, tries to become a proxy from where services can interact with
eduGAIN. Through the use of CYCLONE, we can have multiple services using eduGAIN's features while only having to maintain a single registration towards the eduGAIN federation. This simplifies the deployment of authenticated services in terms of fastness and complexity.

2.1.2 CYCLONE Components

CYCLONE, in turn, consists on some web services deployed by the previously described orchestrators:

- Keycloak: provides the authentication and authorization via OpenID Connect and SAML 2.0to the platform and allows the configuration and management of users and resources via a web interface. The major characteristics and features of this software are described in Section 3.5.

- SimpleSAMLphp: proxies SAML requests from Keycloak towards Identity Providers of the eduGAIN federation. It allows to synchronize the Identity Providers metadata list via eduGAIN's JSON API, so CYCLONE is updated with the latest list of members. The major characteristics and features of this software are described in Section 3.6.

- Cache Clear: clears the user data from CYCLONE after the user session ends. This increases the data privacy and avoids storing user data when is not being used by any other services. Thanks to this CYCLONE can conform to some strict data privacy regulations.

These services are deployed using the CYCLONE platform orchestrators inside a VM located in the private cloud of the institutions part of the CYCLONE federation. Inside the Virtual Machine, Docker is used in order to control this web components and update them with newer versions. Docker provides a simple way to setup the interconnections between the services and also to deployment. Through the use of `docker compose`, Docker can setup the whole environment needed with minimal configuration and just a couple of commands.

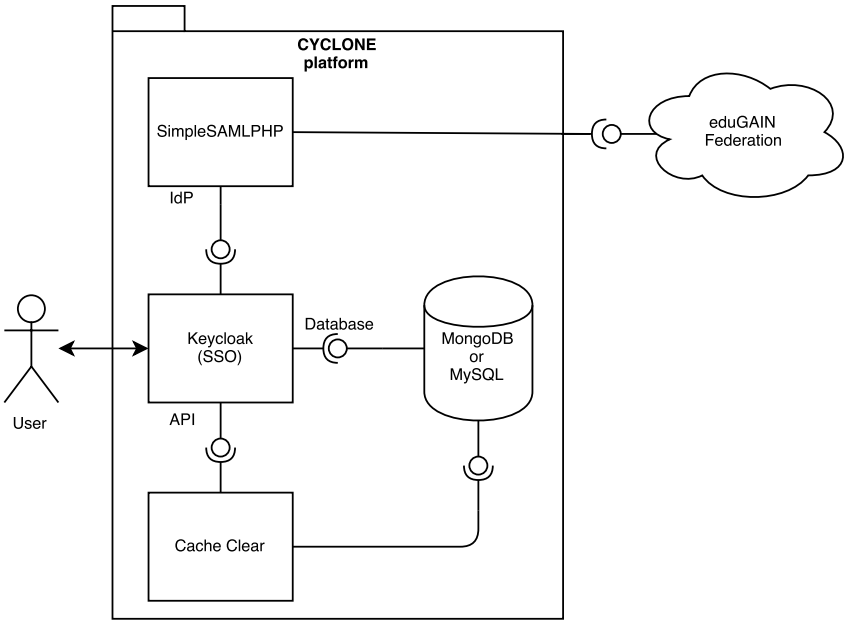

Figure 2: UML component diagram representing the different components in CYCLONE and the dependencies they have.

2.2 Use Cases

This thesis bases its implementation in a set of use cases provided by CYCLONE partners. This section describes the background and characteristics of each of the use cases and how CYCLONE could be a solution for their troubles. Here we summarize part of CYCLONE use cases, described in [6] and [7]. Note that this whole section contains extracts and is heavily based on the work done by Domenico Gallico et al. in [7].

2.2.1 Bioinformatics Use Case

Bioinformatics are concerned with the problem of having huge amounts of data, usually proceeding from DNA sequencers or laboratory metrics. Actual modern sequencers can generate terabytes of data; and together with their relative low price, bioinformatic research teams can easily end with huge amounts of data.

According to CYCLONE's use cases paper, "Bioinformatics community characterizes on using many different software programs to process and analyze the data. It is a common practice to use custom environments with their own dependencies to setup this software and as a result there is a huge fragmentation in the software field, otherwise incompatibility between dependencies and resources might be a problem. Because of this issue, the best approach for bioinformatics scientists has been to use cloud computing, which provides isolation between the different applications and environments and covers the described needs" [6].

The French Institute of Bioinformatics (IFB) consists of 32 bioinformatics platforms (PF) grouped into 6 regional centers found throughout the entire French

territory. The IFB shares a national hub, the "UMS 3601–IFB-core" (also called IFB-core), which is the representative of CNRS in this project. The IFB maintains a shared cloud infrastructure in the IFB-core though the future aim is to design a federated cloud throughout the different nodes of the research network.

The IFB has deployed a cloud infrastructure on its own premises at IFB-core and aims to deploy a federated cloud infrastructure over the regional PFs. This cloud infrastructure is devoted to the French life science community, research and industry, with services for the management and analysis of life science data.

CYCLONE is based on two bioinformatics use cases:

1. *UC1 - Secured Human Genome data sequencing*: Thanks to the decrease in the pricing of genome processing, this kind of analysis has become a common diagnostic practice. Today, according to the CYCLONE use case, "genome analysis is usually realized in the exome, which represents a 5% of the whole genome. With this amount of data, we can't really have problems with revealing private or unique data of a specific human, because there is still a 95% of the human genome without processing, where most of the data is 'stored'" [6].

   However, in the future the objective is to process a full genome sequencing, which in turn would then needed to be handled by drastic data privacy regulations. Through this data, we could envoy all the details of a unique person and as a result the environment where this data is stored and how this data is shared should be ensured to be as reliable as possible in terms of security.

2. *UC2 - Automatizing Microbial Genomes Analysis*: Thanks to advancements in genetics analysis, such as Next Generation Sequencing (NGS), genome sequencing costs have decreased. As a result, scientists now perform analysis in large collections of related genomes (strains), rather than in individual genomes. Even though the cost has decreased, the analysis still take a long time to be finished, and scientists spend more time in sequencing the data rather than analyzing it.

   Thus, there is a need on automatizing the process so scientists can focus in the most important step: the post analysis of the sequenced genomes. IF-MIGALE, one of the bioinformatics platforms in IFB, created an environment to analyze the synteny (conservation of the gene order along the genomes) and to store the results into a database. This environment includes a web interface allows interacting with the analysis and to setup the configuration settings of the environment.

   To set up this environment, create the required database and scale up the resources (this process has high computing resources consumptions), it is needed advanced system administration knowledge. In this context, bioinformatics scientists would have a steep learning curve just to deploy a commonly used cluster of servers. CYCLONE cloud federation aims to simplify the setup of this kind environment into something as easy as just one click deployment with the ability to choose the preferred cloud to where to deploy it.

2.2.2 Energy Use Case

Recently, Germany has approved a new law, the "Energiewende", which consists in a set of climate change mitigation goals that aim to replace the usage of fossil resources with sustainable, green energy sources. This set of changes in law are focused to omply with the European Union's new "202020" legislation.

This means that most of the energy has to come from distributed origins such as solar panels and wind turbines rather than from centralized systems such as thermal or nuclear plants. This new grid of Distributed Energy Resources (DER) need to be monitored and controlled so there are no changes in the energy supply even if the amount of wind and sun variate. For this to happen, this grid needs to become smarter and needs integration with the latest ICT and Big Data technologies, otherwise the system's efficiency might decrease without any warning.

Another concern, is that this DER have different characteristics in power generation, location and amount, which eventually redefines the grid into more complex heterogeneous system. Thus, for easier management, different parts of the grid are aggregated in groups of distributed energy resources called Virtual Power Plants (VPP). This provides s single management point for this group of resources. A VPP and generates raw data on the usage and generation values and thanks to the fact that the grid's global governor system can coordinate different VPPs to cover the needs of the grid.

The data generated by the system, will be available for applications in the energy economy field such as energy brokering or dynamic pricing setting based on the energy usage.

Because of the complexity of the system, it is needed different people with different roles to take responsibility and to manage the different parts of the grid.

- DER Owner: the owner of the resource. He holds the contract with the energy grid, thus personal data of the DER Owner is stored in a secure database in order to contact him in case of problems or incidences. He can authorize other services to access his data for other matters. Also he can see an overview of basic usage of his DER.

- DER Operator: the manager of the resource in terms of configuration, and review of its status. It can be as well the DER Owner. He has access to the status, and historical data of the DER, plus identification data of the resource, so it can provide maintenance in case of technical failure.

- VPP Operator: the manager of a Virtual Power Plant. He can see the different details of the DER resources contained inside the VPP. His task is to manage, control and overview the energy generation and consumption under an specific VPP.

All three roles should have a web interface to access the data both from a computer or a mobile phone.

CYCLONE is based on one energy use case:

1. *UC3 - Virtual Power Plant*: As described previously, a VPP integrates different distributed energy resources (DER) and combines them into one power plant. Given the sensitivity of the data managed by a VPP, the platform managing the energy grid needs to be properly secured in both data storage and data transmission cases. Also, the platform needs to be deployed into a distributed cloud environment, in order to support the different applications making use of the platform. All together access to the different resources need to be secured via authentication and authorization access control.

2.3 User requirements

Through the different use cases previously exposed, we can detect that there are common requirements:

- *R1*: Need to be secure as they contain sensitive data from both the users accessing the system and data being processed (sequenced genomes and energy usage and generation data)
- *R2*: Needs to have multiple interfaces that must be secured via authentication and authorization: a web interface, an SSH interface and possibly a remote desktop interface via X11.
- *R3*: Needs to run in an internal isolated network.
- *R4*: May need to be deployed across multiple clouds. The difficulty to deploy a clone of the platform should, thus, be simple and run in any configuration of the cloud or VM.
- *R4*: Environment and VPP deployments need to be done dynamically and in an accessible fashion. The procedure to deploy new nodes in the network has to be as simple as a button click if possible.
- R5: Needs to support a federation of different entities and individuals. Both use cases rely on a user federation where different groups of people and resources associate together. This results on a need to regulate the free access to data and accept some agreed rules when using the sensitive data.

We can see a diagram depicting the generic usage which defines most of this requirements in Figure 3.

2.4 Stakeholders

In the previous section, we described the needs of the users, which are the scientists and energy network managers. However, there are 3rd parties which are also involved in the requirements of CYCLONE. They can be for example national legislation, rules imposed by the national or local education and research network or any other entity that is related to the privacy and user data usage legislation.

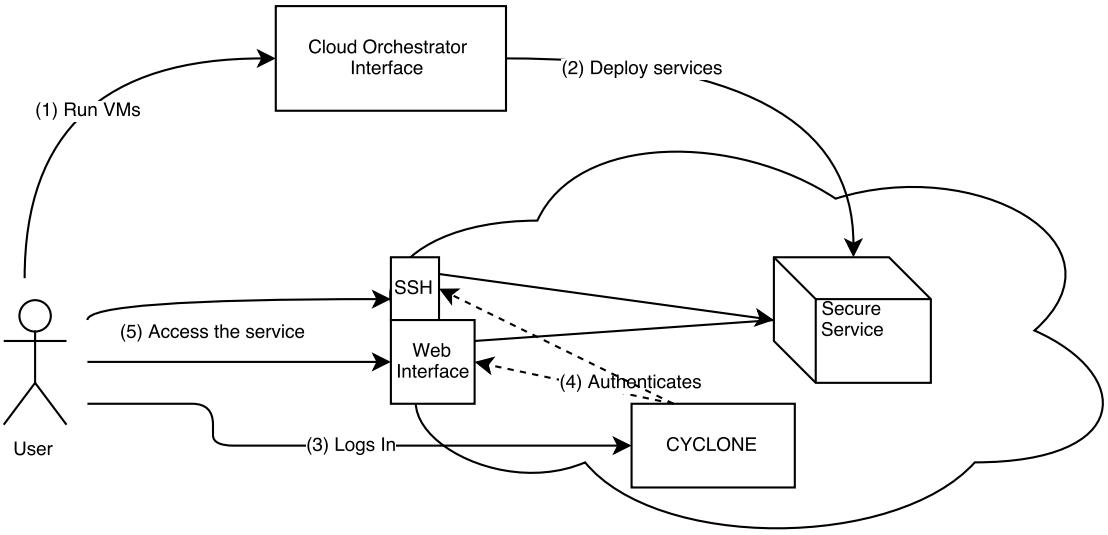

Figure 3: Diagram representing a generic structure defined by the use cases.

1. From TU Berlin: following the data protection rules of the university, information that we gather from the authentication service such as mails, names and similar cannot be stored in any server. It is only allowed to store data meanwhile the user has a session open in CYCLONE. This was the main reason to create the cache cleaner component, which clears the data of users who's session is expired.

2. From SURFnet, in The Netherlands: their requirement to use CYCLONE is that CYCLONE identity provider needs to have a minimum of A rating when validating the connection security with SSLLabs SSL test tool. Otherwise, they treat the service as insecure.

3. eduGAIN: CYCLONE needs to fulfill the different requirements of the eduGAIN federation in order to be part of it make use if its services. This requires having a SAML endpoint and approving a set of contracts proofing that you will not misuse the data of the federation.

Thus, in order for CYCLONE to be used by most of potential users, we need to cover this requirements required by them.

3 Background

This chapter explains the basic knowledge needed on Single Sign-On and related technologies such as SAML 2.0 and OpenID Connect needed to implement this project. First we define the terms 'authentication' and 'authorization' and compare their meanings and their usages. Then we explain what are Single Sign-On systems and finally we describe two SSO systems which are being used by the Keycloak platform (described in below subsection 3.5) in this thesis' implementation.

3.1 Authentication and Authorization

In any secured system there are two processes that are used in order to determine if a protected resource is allowed to be accessed: authentication and authorization.

Authentication is the process to determine if a user claiming to be himself is really saying the truth.

In the other hand, authorization is the process of granting someone or something to access protected information, resources, data or services. The one requesting the access does not have to be the owner of the protected information, and can request access in behalf of the owner of the data. In order to determine the positive or negative authorization, the services may need access to attributes of the user credentials so the privileges of the user can be determined and thus process the authorization.

Authorization heavily depends on authentication in order to provide a proper authorization. If we have not properly validated the credentials of the service requesting the authorization, we may be allowing access to restricted data to undesired users, and thus rendering the whole security system useless.

Both terms meanings are usually confused between each other, even if they define different concepts.

Authentication and authorization are usually processed per domain to protect the user against CORS attacks [8], this means that for each web service or resource that the user wants to access, the user needs to be both authenticated and authorized. Of course, this would make the user to enter their credentials several times, which would provide a bad user experience. In order to mitigate this problem, there are the Single Sign-On technologies.

3.2 Single Sign-On

Single Sign-On [9] is a set of methods which model systems that allow to authenticate users in multiple environments in a secure and easy way. Their aim is to allow the user access to multiple computing domains while authenticating himself only once. These systems do not only provide a friendly access to the user, but also allow administrators to update authentication and access information for multiple web services at once. As a result, it avoids users having to authenticate themselves in each page, and simplifies the management for the domain administrators as the credentials management is not distributed between multiple sites.

3.2.1 SSO generic architecture

SSO systems scenarios usually consist on three main components [10]: a trusted third party called Identity Provider (IdP), a user agent (UA) who is trying to authenticate himself and a web service provider (SP) that is requiring authentication.

UA and SP will not share any private information as UA will not authenticate towards SP but to the IdP instead. IdP is trusted by both UA and SP, which means that UA has a mean to authenticate himself towards the IdP and the SP trust validated data issued by the IdP.

Web services act as Clients of the IdP. This Clients usually need to have credentials against the IdP to demonstrate that they are authorized to receive the user's verified sensitive and private data. With this validated data, clients can take a final decision in authorization to decide if a user can access or not the protected resource for which they need the authentication.

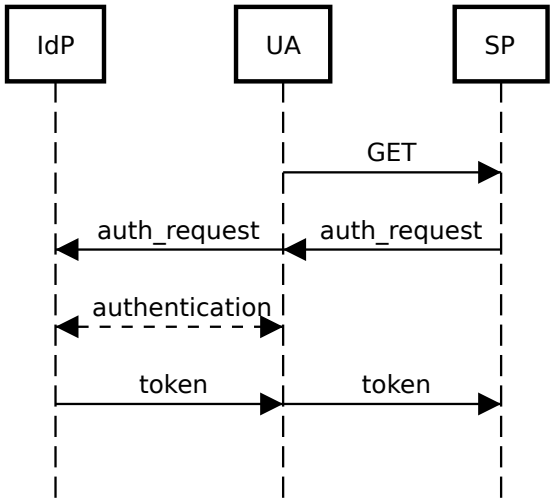

Figure 4: Diagram representing a generic overview of an SSO authentication workflow

Each SSO system has its own authentication workflow, however, most of them share a generic protocol described in Figure 4:

1. The UA requests to access the SP.

2. The SP redirects the UA to the IdP to start the authentication. This redirection contains an *Authentication Request*, which includes characteristics from the SP, such as who is he, which encryption does he accept and similar. This data will help IdP determine to return the needed validated data and its format that SP needs.

3. With the redirection, UA starts a strong secured connection with the IdP. A secure connection is needed as the UA will have to share private credentials to authenticate with the IdP.

4. UA starts the authentication with IdP. It can be done via username and password, but it can also be done through other kind of data such as One-Time Passwords, redirecting to another IdP, or using a cookie from UA's browser to recall a previous authentication.

5. After authentication, IdP issues a token to UA and redirects him back to the SP who originally did the request. This token should contain some information of the UA, and needs to be integrity protected.

6. UA delivers to SP the token, who can validate its integrity using public keys available from the SP. Also, he can use UA's data included in the token in order to take a decision on allowing or not UA to access the protected resource.

This is an example of generic workflow, however, SSOs can have more than one workflow, allowing different authentication methods which can provide more or less restrictions depending on the trust relationship with the user and/or the client.

In order to extend the authentication with authorization, there are a couple of extra steps to be done:

- When authenticating, depending on the trust relationship between the IdP and the SP, UA might to accept an authorization with which it allows the SP to access some of its data and private resources.

- The token becomes a set of tokens. It has and *Access Token*, which is really short lived and allows to request access in behalf of UA, and a *Refresh Token*, which has longer TTL and allows to request new Access Tokens. This tokens need to be stored securely as it they are the keys that allow access to the private data of UA.

- With this tokens, IdP can request to SP extra UA's identity data not included in the Access Token, such as the address, birth date or any other attributes that IdP might have from the UA.

There are many SSO frameworks which provide this kind of solutions and are widely used, for example CAS [3], OpenID [11], OAuth [12], Facebook Connect [13] or the Security Assertion Markup Language (SAML) [14]. Some of them are SSOs which only provide authentication OR authorization, they don't provide both processes. It's important to take this in account when deciding which framework to be used.

In this thesis we will focus in two of this SSO architectures: OpenID Connect 1.0 and
SAML 2.0.

3.3 OpenID Connect 1.0 and OAuth 2.0

OpenID Connect (OIDC) [3] is the third generation of the OpenID technology. OpenID Connect is an evolution of the OpenID 1.0 and 2.0 standards. These last two protocols provide both provide authentication but no authorization. Nowadays we also need authorization to take a decision on giving access or not. OIDC implements

an evolution of the previous OpenID standards into a new version that includes both authentication and authorization.

OpenID Connect 1.0 is in general an extension of the extensively used OAuth2.0 protocol, which provides authentication on top of OAuth2.0 authorization thanks to an inspired validation technology based on the OpenID 2.0 protocol. It allows to SP to verify the credentials of UA and then perform authorization using this validated credentials, as well as to obtain user attributes through a REST-like API. With this, OIDC is fully compatible with the OAuth2.0 standard. However, it is not compatible with OpenID 2.0 or other previous versions of the OpenID protocol.

3.3.1 OAuth 2.0: Authentication

OAuth2.0 [15] is one of the most extensively used authorization frameworks today. OAuth2.0 consists on next evolution of the OAuth 1.0 protocol specified by the IETF in RFCs 6749 and 6750 (published in 2012) which allows multiple workflows of authorization depending on the trust relationship between the IdP and the SP (called Clients in OAuth terms). However, people have been mistaking this technology as an authorization service rather than authentication service. Some implementations of OAuth2.0, like the one created by Facebook and rebranded as Facebook Connect, include an Identification Token on top of the OAuth2.0 protocol, in order to provide an authentication layer for OAuth2.0.

Other options to fix the authentication not included in OAuth problem are using the different SSO technologies that provide authentication in conjunction with OAuth2.0. Example of possible authentication frameworks would be OpenID 1.0, JASIG CAS. The objective is to provide a token, with which OAuth2.0 can validate the credentials of the people requesting the authentication.

Through the use of OAuth2.0 as a base, OIDC has compatibility with most of the implementations of OAuth2.0 and thus simplifies an upgrade from OAuth2.0 to OIDC.

3.3.2 OIDC Specifications

The OpenID Connect 1.0 specification consists of many components [16]. The most relevant ones from all the specifications are:

- Core [17] – Defines the core OpenID Connect functionality: authentication built on top of OAuth 2.0 and the use of Claims to communicate information about the End-User.
- Discovery – (Optional) Defines endpoints from where clients can dynamically fetch configuration from OpenID Providers.
- Dynamic Registration [18] – (Optional) Defines how clients can register dynamically to use an OpenID Provider.
- OAuth 2.0 Multiple Response Types – Defines several specific new OAuth 2.0 response types.

- OAuth 2.0 Form Post Response Mode – (Optional) Defines how to return OAuth 2.0 Authorization Response parameters using HTML form values that are auto-submitted by the User Agent using HTTP POST.

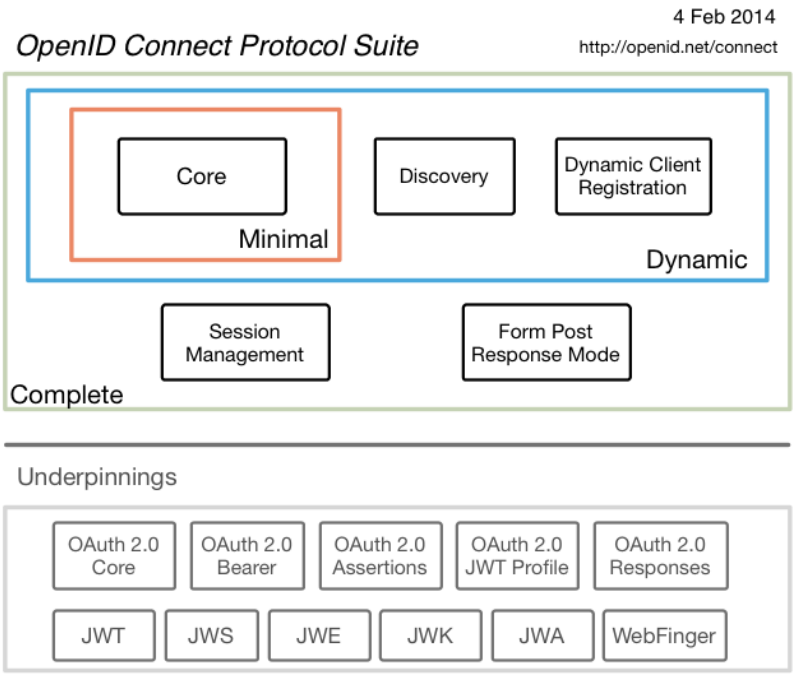

Figure 5: Diagram representing the different specifications conforming the OIDC suite (from http://openid.net/connect/

One of the optional specifications of OIDC is Dynamic Client Registration [18], which allows to register a new client into the IdP automatically. This registration can be done through the interaction with OIDC's API and the use of JSON Web Tokens (JWT) signature keys [19] or via other means to verify the authenticity of the new client registrant.

Final OpenID Connect specifications were launched on February 26, 2014. So OIDC is a quite recent protocol and is not really extensively used. However, it has high support from big IT companies such as Google, Microsoft, and Deutsche Telekom between others. For example Google uses OIDC in their SSO systems, in parallel to SAML 2.0.

A certification program for OpenID Connect was launched on April 22, 2015. This certification provides assurance that the implementation of the OIDC server has been created according to the specifications of the protocol.

3.3.3 Scopes and Claims

OAuth2.0 defines the claims as the different attributes of the user that the Relying Party can request. They can be, for example the email, phone, address of the user. OIDC provides a set of default claims that should be included in all the OIDC

implementations [17]. Other custom claims can be included to extend the user attributes.

This claims are sorted in scopes. A scope is a group of related claims that can be requested access to by the Relying Party. When the authorization is requested, the request includes a scope. The User Agent needs to approve the access to this claims to allow the Relying Party to access the user attributes.

OIDC contains an specific scope *openid*, which allows the clients to request authorization to access to the user's attributes. This is a feature not included in OAuth2.0 and provided by OIDC allows to manage and authorize the access to the user's data via 3rd party clients.

3.3.4 Authentication Flows

OIDC has multiple authentication and authorization [20][17] inspired in OAuth.

- Authorization Code Flow: According to OpenID specifications, "the Authorization Code Flow returns an Authorization Code to the Client, which can then exchange it for an ID Token and an Access Token directly. This provides the benefit of not exposing any tokens to the User Agent and possibly other malicious applications with access to the User Agent. The Authorization Server can also authenticate the Client before exchanging the Authorization Code for an Access Token. The Authorization Code flow is suitable for Clients that can securely maintain a Client Secret between themselves and the Authorization Server." [17]

- Implicit Flow: According to OpenID specifications, "the Implicit Flow is mainly used by Clients implemented in a browser using a scripting language. The Access Token and ID Token are returned directly to the Client, which may expose them to the End-User and applications that have access to the End-User's User Agent. The Authorization Server does not perform Client Authentication." [17]

- Hybrid Flow: The hybrid flow is a combination of the above two. It allows to request a combination of identity token, access token and code via the front channel using either a fragment encoded redirect or a form post.

3.4 SAML 2.0

According to its specification [14], "The Security Assertion Markup Language (SAML) standard defines a framework for exchanging security information between online business partners. It was developed by the Security Services Technical Committee (SSTC) of the standards organization OASIS (the Organization for the Advancement of Structured Information Standards)".

SAML 2.0 is based on XML and uses security tokens with assertions to share information about a principal (usually a user), between a SAML authority and a consumer of this data. Its architecture allows web based authentication. It was

Table 1: OpenID Connect Workflows Comparison

Property	**Authorization Code Flow**	**Implicit Flow**	**Hybrid Flow**
All tokens returned from Authorization Endpoint	no	yes	no
All tokens returned from Token Endpoint	yes	no	no
Tokens not revealed to User Agent	yes	no	no
Client can be authenticated	yes	no	yes
Refresh Token possible	yes	no	yes
Communication in one round trip	no	yes	no
Most communication server-to-server	yes	no	varies

approved as standard in March 2005 and replaced SAML 1.1. SAML 2.0 is the merge of three different technologies: SAML 1.1, Liberty ID-FF 1.2 and Shibboleth 1.3.

SAML is extensively used by many companies and institutions and is a dominant SSO technology through many internet web services. This allows interoperability between different different companies SSO systems as they are using the same SAML protocol. As an example, eduGAIN defines as a requirement having a SAML 2.0 endpoint in order to be part of the federation.

SAML 2.0 is pretty similar to OIDC in terms on how it works [21]. It uses the redirection of the user between the Service Provider and the Identity Provider in order to transfer tokens and validated and signed XML assertions.

3.5 Keycloak

Keycloak [22] is an open source Identity and Access Management solution created by RedHat. Its aim is to provide a simple platform to secure and protect web services with just basic configuration. It runs on top of Jboss EAP application server (now known as WildFly), and uses Java as programming language. As a database backend it can work with MySQL, MongoDB and PostgreSQL databases.

Administration in Keycloak works through a web control panel where all the settings can be changed. This avoids setting configuration through code and allows a short learning curve. The interface provides an easy overview of users, clients and permissions and explanations and tooltips for each of the settings.

Keycloak works with realms, which manage a sets of users, clients, roles, groups and its own configuration. Each realm is isolated from each other, and can only manage authenticate users under their control. There is also a *master* realm which provides full control over all the other realms and allows advanced operation settings of the other realms. Users are managed via roles and groups. Roles provide individual

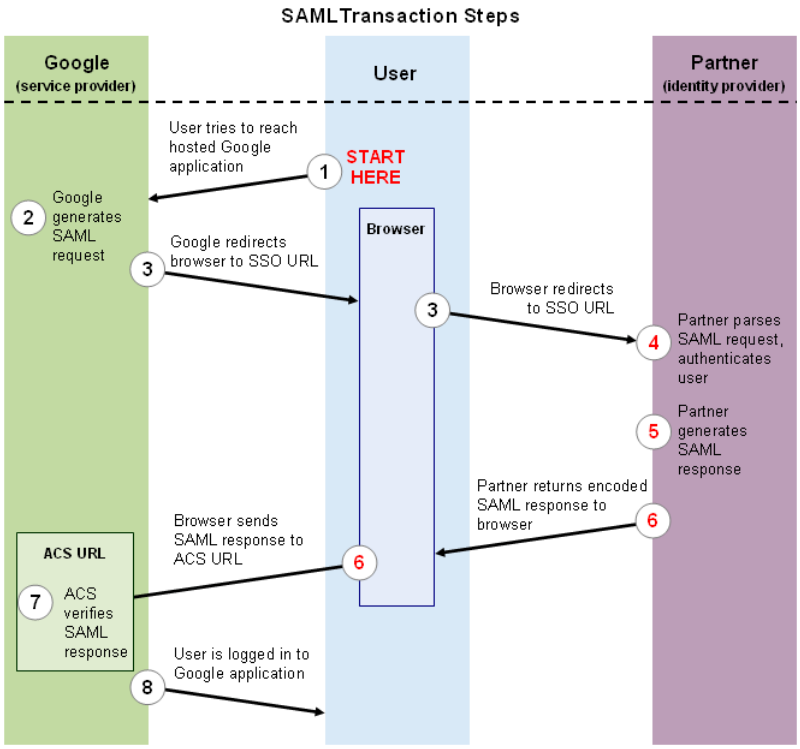

Figure 6: Diagram representing a SAML 2.0 transaction workflow (from : https://developers.google.com/google-apps/sso/saml_workflow_vertical.gif

permissions to each user, and groups organize users in a hierarchy structure. Each group has a set of roles which get assigned to the users inside of the group. Users inherit as well all the roles of the parent groups to which they belong to.

Keycloak provides authentication and authorization through two SSO protocols: OpenID Connect 1.0 and SAML 2.0 [23], described in previous subsections 3.3 and 3.4. For each client created in a realm, you can choose to use either OIDC or SAML as the SSO protocol. Keycloak also allows client templates, to use as a base to create new clients. For each individual client we can setup fine grained authorization to resources, client specific roles and the scope of claims we want to allow the client to access.

3rd party Identity Providers can be configured easily, so new users can register or identify with social networks accounts such as Facebook, Google, Twitter, etc. As an alternative it also allows to use generic SAML or OIDC providers providing the custom configuration for the Identity Provider.

Though Keycloak is designed to cover most use-cases without the need of coidng, Keycloak can be extended and customized via Service Provider Interfaces (SPI) modules. SPIs provide different functions to Keycloak and extend its functionality. Keycloak provides in its core two kinds of SPIs: private and public ones. Private SPIs are part of the core and can't be extended. Public SPIs can be extended and used by other SPIs.

Keycloak's configuration can be exported in XML format. This allows to keep con-

figuration changes that are stored usually in the database and to share configurations with other instances.

Currently, Keycloak is in version 2.5.0 and in full development. In their website there isn't a clear roadmap of the future features that will be implemented, though discussions and decisions are taken in their developers mailing list.

Also, Keycloak has been certified as a valid OpenID Connect implementation, which means that this platform conforms all the requirements of the OIDC protocol.

3.6 SimpleSAMLphp

SimpleSAMLphp [24] is a native implementation of SAML 2.0 written in PHP which can work both as a Identity Provider or Service Provider. The choice to use SimpleSAMLphp for CYCLONE comes from the need to synchronize the platform's data with eduGAIN's metadata services. Keycloak, even if you can use a SAML 2.0 endpoint as Identity Provider, per itself it doesn not provide a simple way to synchronize SAML metadata with a service such as the one provided by eduGAIN. In order to Keycloak support, a custom SPI extension would be needed to be developed. This is a rather tedious implementation.

Instead, a simpler solution is to use SimpleSAMLphp as an aggregation proxy of the IdPs towards Keycloak. SimpleSAMLphp supports the dynamic updates of metadatada provided from eduGAIN, and allows users to select with which of eduGAIN's members he wants to authenticate. Then, the authentication information is relayed towards Keycloak and can be authenticated in the underlying services. This means that the endpoint to be registered in eduGAIN's metadata directory needs to be SimpleSAMLphp's one as it will be the one consuming the users data provided by the federation.

Though SimpleSAMLphp may look a different site than Keycloak, using a common theme, allows the users to trust that both sites behave as a single web component.

4 The service registration API

In subsection 2.3 we described the different requirements that need to be covered so CYCLONE can provide a complete solution. Specifically requirements R3 and R4 define that users should be able to deploy dynamically their services. This services need client credentials so they can authenticate users against the SSO server and fetch their credentials and private data.

In this chapter we first analyze the state of the art and the different options that Keycloak provides out-of-the-box. Using this as a basis, we defend creating our own solution as these ready made ones do not cover the needs described by our use cases. Next, we compare the different options to create our own Keycloak extension, and finally we provide a detailed description on how we have implemented a Keycloak SPI module to extend the platform with the required registration functions.

4.1 State of the Art

In Keycloak clients are usually created in a manual fashion via the web interface where an admin will create the credentials and forward them to the requesting person. This procedure, though, is quite slow and requires user interaction, something not fitting when using automatic deployments through a deployment orchestrator. For CYCLONE to work, there is a need of an automatized system to create clients.

In actual existing use cases of SSO implementations, such as the ones provided by Facebook Connect or Google Accounts, this kind of credentials can be requested via a web interface and even using an API. The credentials are then tied not only to a client but also to the person requesting them and allowing thus to identify who has the ownership and rights on the dynamically created client credentials.

Keycloak, the SSO platform used by Cyclone, integrates some mechanisms to create the clients and its credentials:

- *Using Keycloak API*: Through the API that Keycloak exposes, new clients can be created. Access to the API can be obtained requesting an access token in the admin API's token endpoint using user's credentials as authentication This access token can then be used to interact against the API itself.

  In order to create a client in the API, the requesting user has to have a "create-clients" role in the specific realm. However, to edit any of the client settings a user then needs a "manage-clients" role. The combination of having this two roles created by Keycloak grants permissions not only to create but also edit and update any client in the realm. Giving these roles to many people would result in an easily exposed client management, and users with these roles would not only be able to edit their own configuration but also configuration relevant to other clients and services. A user with this rights would be able to give all the admin roles to the client's user and then from a client escalate to have superadmin rights. Thus this system should only be allowed to be used to sysadmins or people with similar privileges. Another option would be to allow only creation access to users, but that means that any change that needs to

be done to the client (set a consent, scope of users, etc.) needs to be changed manually by someone who has the proper rights, usually an administrator.

- *Using OpenID Connect Client Registration*: When planning the implementation of this thesis, Keycloak in version 2.0 provided an API endpoint implementing the OpenID Connect Registration specifications defined by the protocol.

  The OIDC registration endpoint works using shared registration tokens, which anonymous people can use in order to register clients by themselves. This registration tokens have a limited amount of uses and defined lifetime – usually long – and anyone who has this registration token can register a new client. The use of this tokens is focused to anonymous non authenticated users. As a result, there is no way to track who exactly used the token and who is creating clients. At most the only possibility would be tracking from which IP has the registration been done. Also, even if the token has long expiration time, it can still end under ownership of a malicious attacker, who could abuse it to fetch private data from users.

  With this registration token, a new client can be created, and after the successful request a new token is received. This second token is a "client manager token" which can be used to update settings of the client. However, as per Keycloak 2.0 specifications it also allows to edit any other client aside of the created one. This means that any one who can create a client can eventually edit any other client in the same realm. Not only that, but this "client manager token" needs to be kept safe somewhere by the client creator in case it is needed to update the client. Also, after each of the usages of this token to update the client settings a new "client manager token" will be given replacing the previous one. So the creator of the client needs to keep safe a token that will get updated from time to time and that has excessive privileges.

  This structure is as it is because as explained previously there is no way to track who is the people who is using the tokens and thus all this tokens need to be also anonymously used.

Summarizing, there are two existing solutions to create clients but both of them provide or require excessive permissions to the users using them. This excess of permission could eventually cause security issues and possible user data leaking. According to Keycloak's developer mailing list there is a plan to implement more granular permissions in the platform, but there is no final decision on when this feature will be implemented. Thus, a new solution needs to be implemented that includes fine grained permissions when creating and managing clients and that tracks who is creating or using this rights.

4.2 Initial Plan

A new solution that allows fine grained permission and that allows us to track who is registering the clients needs to be implemented as the existing solutions that

Keycloak provides do not suit the user needs. The objective of our implementation is to have two main characteristics:

- The client registration API should allow us to authenticate ourselves using our SSO credentials. As it is needed to know who is registering clients.
- Only people with the required privileges should be authorized to create and edit clients, disallowing to take control of any other client. As we would have a proper authenticated user, we can define per-user authorization for each client. In the end we would create owner and resource relationship between a user and client respectively.

Following this two objectives, there are two main different approaches that we could use to implement our requirements into Keycloak:

1. Create a wrapper for the existing Keycloak APIs

   The first approach would be create a new API that uses the existing APIs as a backend and that extends them to provide the requirements we need. For example, we could create a micro service compoenent that would allow us to authenticate using our SSO credentials, and then it would relay the creation of the client against the OIDC client registration endpoint. This kind of implementation is less invasive and as it relies on web APIs it is easier that in the future there are no breaking changes between Keycloak and the extension.

   The main drawback of this approach is that we are creating new relationships and we would need to store this data somewhere, preferably in a database, not the same as the one being used by Keycloak, which would add a new component into the platform. This increments the number of components of the CYCLONE platform, and also decentralizes the authentication and SSO data between databases and services, which in turn rises the complexity and the maintainability of the the whole set of components.

2. Create a new API that fulfills all the requirements using Keycloak's SPI extensions

   This second approach would consist in implementing a whole new API endpoint, integrated directly into Keycloak's using its SPI module system [25]. As a result we have to interact with Keycloak's core SPI modules to manage the clients. This makes this implementation more fragile, as internal APIs are more likely to be changed. However, it also centralized and unifies all the new logic inside of Keycloak's container, and thus simplifying the deployment and maintainability of the system. Also, it is a more efficient approach in terms of resources as DB queries can be tailored to the needs of the logic. Other benefits would be that there are more methods exposed internally that allow more flexibility when interacting with the client and user entities.

Another parallel discussion is about a possible interface that could be created to manage the API. Creating a the API using a wrapper, would mean creating a new

external component, outside of Keycloak and thus its UI. This results on having two different UIs where inputting the data, and also increasing the complexity of the system. In the case of integrating the solution as an SPI inside Keycloak, this allows us to (if needed) integrate the UI into Keycloak's UI. This can be achieved through extending Keycloak's theme in a plugin and exposing the API internally into the web interface. This, of course would also increase the security as we are relying in the secure implementation of Keycloak's UI and we do not have to proof the security of a new website from scratch as it would hapeen in the first case.

Out of this two options, our decision was to implement a new SPI in Keycloak. The main driver was the maintainability of the system, and avoiding to increase the complexity with more components. Also, integrating the system directly into Keycloak would allow us to use similar authentication and security systems as the other API endpoints are using.

4.3 Architecture

To create our own extension module for Keycloak we have inspired ourselves in Keycloak's "domain-extension" [1] example. This example contains guidelines to create an authenticated endpoint and how to integrate calls to the database from within it. SPI extensions can contain multiple providers inside themselves and use already existing providers that are part of Keycloak's core. Service Providers Interfaces (SPIs) in Keycloak are written in Java and make use of the factory method pattern to create new providers. This factory pattern "provides an interface with which a client can obtain instances of classes conforming to a particular interface or protocol without having to know precisely what class they are obtaining" [26]. Thus, through the use of this pattern, Keycloak can instantiate the new classes that extend their own functions.

To implement this pattern in Keycloak, we need to make use of the *ProviderFactory* and *Provider* interfaces. The ProviderFactory interface defines the factory class and includes the logic to create a new object of an specific class, which implements the Provider interface. In the other hand, the Provider interface defines a Provider object, which when extended, contains all the possible methods that we can use to interact with the provider. This interface also exposes inner methods from where we can request and intereact with other SPIs.

Also, a Keycloak SPI extension requires a class extending the *Spi* interface, where we can define the basic parameters and name of our SPI.

In our SPI we need to at least implement three different Providers, that will cover the different needs we have. This providers are located in their own sub-packages inside of the *org.cyclone.clientowner* package:

- REST API (*org.cyclone.clientowner.rest*): defines all the API endpoints, executes authentication and authorization and forwards the requests to the proper methods of the resource logic.

[1] https://github.com/keycloak/keycloak/tree/master/examples/providers/domain-extension

- Resource logic (*org.cyclone.clientowner.spi*): defines all the main logic of the SPI, and calls the methods in JPA entities to update them and update the database values. It also interacts with other SPIs such as the user, clients and realms SPI to take decisions in the logic.

- JPA Database Entity (*org.cyclone.clientowner.jpa*): defines a JPA database entity interface and how the objects are stored into the database. Principally it defines the model and the relationship with other items in the database.

The base of our extension is to create an ownership link between the a client and a user entities. Also, we need to include the realm entity so a relationship cannot be used outside of a specific realm. In this relationship we can save the information which indicates which user owns an specific client in a per realm basis. To create this design we need to use the client, users and realm SPIs, so we can access the respective database entities of each provider.

Finally, the JSON models used in our implementation are the same as the ones used by Keycloak Admin REST API [2]. The reason is that doing so we keep the same object formats for all the different requests towards Keycloak. This also simplifies later on the translation from JSON object towards internal database entity object.

4.3.1 Authentication

To authenticate, using the 'domain-extension' example as a reference, we can use the same authentication system as we have in Keycloak's administration API. This can be done through the AppAuthManager class, which provides us functions to validate a bearer token given a context. The bearer token can be obtained through authentication using JWT or client credentials against Keycloak's OIDC token endpoint, which its path is `/auth/realms/master/protocol/openid-connect/token`. In general the workflow to authenticate against our endpoints is based on the Authorization Code Flow described in subsection 3.3.4:

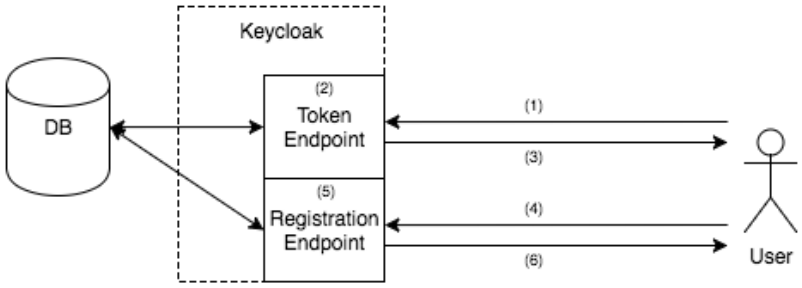

Figure 7: Represents the workflow required by the registration API implementation to register or modify a client

[2] http://www.keycloak.org/docs/rest-api/

1. The user authenticates with its credentials against the OIDC Token Endpoint. Authentication can be done through JWT or client_id and client_secret credentials.

2. Keycloak generates OIDC tokens (access and refresh tokens), which allows the user to authenticate in other endpoints.

3. User gets these access and refresh JWTtokens.

4. User sends the registration data together with a Bearer token (access token) to authenticate. Registration data is a JSON with format ClientRepresentation.

5. Extension logic.

6. The user gets redirected to the new created resource (with all the information of the new client), or returns the proper code depending of the REST action.

Then, an example cURL POST call to create a new set of tokens would consist in a request with the following header and body:

```
curl -X POST
-H "Accept-Language: application/json"
-H "Content-Type: application/x-www-form-urlencoded"
-H "Cache-Control: no-cache"
-d 'grant_type=password&username=admin&password=admin&
    client_id=admin-cli' "http://localhost:8080/auth/
    realms/master/protocol/openid-connect/token"
```

Listing 1: cUrl command to generate a new set of OIDC tokens

In this case we are doing a password authentication, using the client_id and the client_secret. However, we could also authenticate ourselves using JWT authentication. As the generated access tokens contain encoded data from the user, we later figure out who is who when requesting access to our implemented endpoints.

Then, afterwards, we can do a call with the provided access token to the wanted endpoint. This access token will last usually around 60 seconds. This is a configurable setting in Keycloak's admin interface. In case of expiration, a new token can be requested through the token endpoint using the refresh token also obtained when authenticating against the OIDC token endpoint the first time.

```
curl -X POST -H "Authorization: bearer <access token
    here>"
-H "Content-Type: application/json"
-H "Cache-Control: no-cache"
-d '{
\\ body of the call
}' "http://localhost:8080/auth/realms/master/client-
    registration/"
```

Listing 2: cUrl command to create a new client authenticating with an access token recived from the OIDC token endpoint.

4.4 SPI implementation

In this subsection we describe each of the different packages that conform our implementation – REST API, JPA adapter and SPI logic – and we explain how we implement each of them. Each of the providers that conform the SPI have a FactoryProvider and Provider classes that interface with Keycloak so we can extend with our custom code. A general overview of all the different classes that conform the whole extension can be found in Figure 8.

4.4.1 REST API

To create the REST API for our registration system we use Java API for RESTful Web Services (JAX-RS), which is the framework that Keycloak uses to create its REST endpoints. To do so, first we create a a ResourceProviderFactory and we use

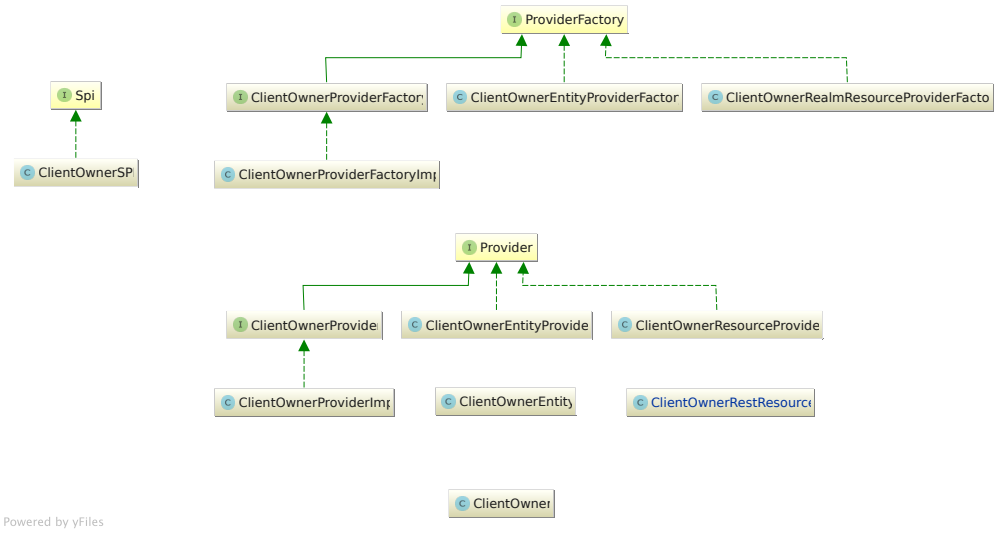

Figure 8: Class relationship diagram depicting the relationship between the different classes that conform the whole registration API extension.

it to create a ResourceProvider object. In the Factory object we define the identifier of our provider, in our case *client-registration*. With this set of classes we indicate to Keyclak, using the factory pattern, that our REST provider is defined in our ClientOwnerResourceProvider class. In turn, the ClientOwnerResource Provider class initializes all the API settings and instantiates a ClientOwnerRestResource object that contains the API. We can see the relationship of these classes in Figure 9.

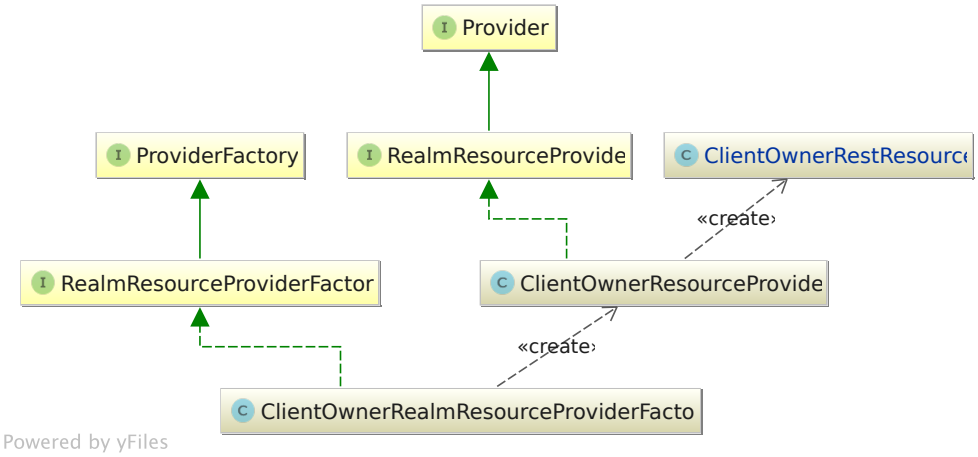

Figure 9: Class relationship diagram depicting the relationship between the different classes that conform the API in the registration API.

Then, we can create the different endpoints using annotations in functions. With the annotations (see lines 7 to 11 in Listing 3), we can tell Java which kind of data we are expecting, which is the path to this endpoint and which is the format of our

answer, which in our case we have set it to JSON. We can also define the path variables in the function's arguments. The name of the function does not affect at all the endpoint. From inside the function we have access to the session information and we can call the different functions that we implement in the SPI logic, which are explained later in subsection 4.4.3.

```
1       /**
2        * Endpoint to update clients
3        * auth/realm/{realm}/client-registration/{clientId}
4        *
5        * @return ClientRepresentation of the new client
6        */
7       @PUT
8       @Path("{clientId}")
9       @Consumes(MediaType.APPLICATION_JSON)
10      @NoCache
11      @Produces(MediaType.APPLICATION_JSON)
12      public Response updateClientOwnerResource(@PathParam("
            clientId") String clientId, ClientRepresentation
            clientRepresentation) {
13          checkRealmAdmin();
14          checkClientOwnership(clientId);
15
16          // Logic here
17      }
```

Listing 3: Example of annotation defining the settings of an endpoint

In all the different endpoints we are executing at first the logic to check the authentication of the user. With it, first we make sure that the user has the needed role, which should be fetched thanks to the included bearer token. Then in cases of edition and deletion, we make sure that a user has ownership over the resource before giving access to manipulate it. To do so, we use the context information provided by Keycloak and we use the AppAuthManager class to authenticate the bearer token. Also, at the end of each endpoint's logic we handle any possible error and we construct the requited HTTP errors to inform properly the user. All the endpoints we have created are under the base path *<domain>/auth/realm/{realm}/* which is set by Keycloak. This path allows us to process the calls individually per realm. We have created 5 different endpoints (see Table 2) to manage all the different CRUD operations that we might need in our extension.

All this endpoints either return or receive an object with structure of ClientRepresenation [3] or an array of this kind of items. None of the attributes is required when creating a new client as and otherwise it will use the default value or randomize it. However we suggest to at least set the *clientId* value for easier management of clients. We have chosen to use this representation because Keycloak exposes methods in other SPIs to convert JSON objects to Keycloak models and vice versa in an easy fashion, which simplifies the implementation. Note that the conversion is to models and not to entities used in the database.

[3] http://www.keycloak.org/docs/rest-api/#_clientrepresentation

Path	HTTP verb	Description
client-registration/	GET	Get an array of all the clients available for the actual user
client-registration/	POST	Create a new client with a user ownership
client-registration/{clientId}/	GET	Get an specific client via its ID
client-registration/{clientId}/	PUT	Edit a client
client-registration/{clientId}/	DELETE	Delete an specific client-owner resource

Table 2: List of registration API endpoints and a description of their implemented functions. The base path for all the endpoints is */auth/realm/{realm}*.

4.4.2 JPA Database Entity

Keycloak uses the Java Persistence API (JPA) to manage the access, persistence and structure of data in the database. In Java instead of having to implement the queries to a database and map the results to objects, we can use frameworks such as JPA or Hibernate to manage the interactions to the data storage engines.

Keycloak requires entities to manage database objects. Through JPA this entities are mapped them to their SQL schemas. Though Keycloak supports MongoDB, we have decided to not to implement the required entity interfaces because of many reasons reasoned in subsection 4.5.

JPA works with entities. Each entity maps to a table in the database and they have a primary key to sort the entries. In Keycloak, string UUIDs of 36 bits are used as keys. Also, we can make use of already existing entities in Keycloak so we can create relationships between tables. In our case, we will create a new entity, CLIENT_OWNER, that will have relationships to the user, realm and client entities. Finally the schema and the required migrations of the database also need to be recorded in a separate XML file.

To implement our entity we firstly have created a EntityProviderFactory and its own EntityProvider. Through this classes Keycloak can detect that we have implemented a new Entity. In the EntityProvider class we have defined the schema migrations that we need to execute in the database. In our case, we need to create a table named CLIENT_OWNER which has a primary key column ID and 3 columns: OWNER, CLIENT and REALM_ID. All this columns are a VARCHAR of length 255 as IDs in Keycloak are strings.

Next, we can use JPA annotations to define the different columns in our entity and map this columns to attributes of our class.

```
1       @Id
2       @Column(
3               name = "ID",
4               length = 36
5       )
```

```
6       @Access(AccessType.PROPERTY)
7       @GeneratedValue(generator="system-uuid")
8       @GenericGenerator(name="system-uuid", strategy = "uuid")
9       protected String id;
10
11      @OneToOne(fetch = FetchType.LAZY)
12      @JoinColumn(name = "OWNER")
13      private UserEntity owner;
14
15      @OneToOne(fetch = FetchType.LAZY)
16      @JoinColumn(name = "CLIENT")
17      private ClientEntity client;
18
19      @Column(name = "REALM_ID", nullable = false)
20      private String realmId;
```

Listing 4: Example of annotation mapping JPA columns to object attributes

To find the related clients and users, we need to find them by their unique ID, which is saved in our table. There is a one-to-many relationship from the clients, users and realms to the client_owner entity. To create this relationships we need to create joined columns. However, this will be a relationship that can only work in one direction, as we cannot edit the entities provided by Keycloak that there is a relation. This means that we can only find a user or a client from the client_owner entity but not viceversa. We could create a migration in the database to enable a two way relationship, but doing so we put in danger the integrity of the database, as we would not be sure if Keycloak has plans to update this specific tables. Also, for this same reason we cannot enable a cascade delete to delete clients or users whenever we delete a relationship, as we would need to create migrations that could conflict with Keycloak's. So, instead of deleting via cascade, we have also decided to not to delete the users or clients when deleting a relationship because we could create users that could be used for other means. Instead our decision is to only delete the relationship, which will eventually forbid the user from edition and viewing access.

Finally, we also need to make complex database queries in an easy fashion. JPA allows us to create named queries with custom parameters, so we can find easily records, for example per realm. The actual implemented queries are *find by realm*, *find by ID*, *find by client*, *find by owner* (user), *find by client and owner* and *delete client owner*. When searching by client and by owner we do not have to provide the ID, as JPA requires the whole entity. This is important as this will be relevant when creating the logic that uses this entity.

4.4.3 SPI Logic

This package includes the main logic of the extension. It relies on the previous implementation of the API and database entities parts. To implement this logic we first have to create a the ProviderFactory and its Provider interface. In the

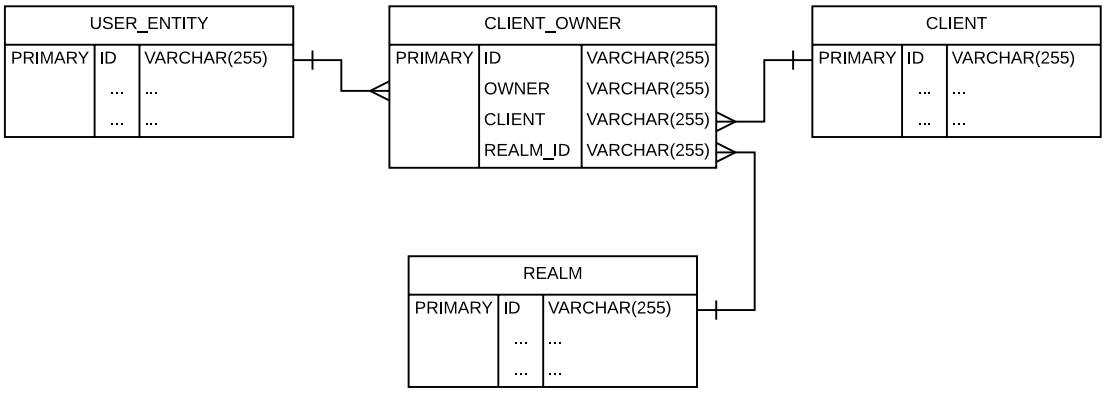

Figure 10: Entity relationship diagram depicting the relationship between the different tables in Keycloak's database. The USER_ENTITY, CLIENT and REALM entities have a relationship of one-to-many to the CLIENT_OWNER entity through the use of their IDs.

Provider interface we need to add all the possible functions that the API can call and that with the logic and database queries we have to fulfill. Then, in the ClientOwnerProviderImpl class we implement all this functions that the interface requires. Implementing this functions using this pattern allows other SPIs to call this functions if needed.

The implemented functions are CRUD functions to manage ClientOwner entities and get/set functions used internally to manage the client and owner relationships of a specific record. This functions we usually receive a model as an argument but need to map it entities. Models do not contain a method to convert to entities, and the JPA database interface requires entities to search some records. Thus, we need to make queries using attributes of the models, such as the ID or the name, in order to find the related clients and users through the use of name queries. Also, we can make use of the EntityManager to execute named queries from core providers, such as users or clients.

Another challenge during the implementation was the fact that sometimes we want to create a client and then use the ID of this client to create a client_owner relationship. In this case, as JPA has not processed the first transaction, we do not have the ID of the client, and thus we have problems when creating records. To avoid this problems we need to forcefully flush the transactions queue

Finally, before returning the results, we have to check that we are not returning too many results or that we unexpectedly have multiple records for the same client_owner relationship. In case we find more than one, we can throw an exception with a message warning about the inconsistency in the database (which should never happen).

4.5 Limitations

Even though we have carefully designed the implementation of this registration API, there are a couple of limitations that need to be reviewed in future implementations

of this software. The first problem is that the actual implementation only works with JPA supported databases which are SQL based databases such as MySQL and Postgres. This means that Mongo, which should be also supported by Keycloak, would not be compatible with out extension. The reason is the database interface of our extension. Keycloak, in order to support MongoDB as a database, needs to have an entity specifically written for this database, aside of the one that we have already created for JPA databases. This would mean reimplementing all the entity management and relationship system of the database. Also, Keycloak does not provide a properly documented procedure on how to implement this MongoDB entity. And finally the Keycloak community is considering removing MongoDB from the set of databases that Keycloak supports [4]. The main reason is maintainability of the software and avoiding deduplication when creating database entities. Because all this reasons, and to avoid implementing a MongoDB interface when it is possible that will be deprecated soon, we decided not to implement it unless is really required.

4.6 Deployment

After implementing our software, we can use Maven Java builder to compile and package our extension. To deploy it to a Keycloak server, we need to use then Keycloak's admin CLI [27]. Through it we can deploy our newly build module, in JAR file package inside the Wildfly server on which Keycloak runs. To do so, we can use the following command:

```
1  /opt/jboss/keycloak/bin/jboss-cli.sh \
2      --command="module add --name=org.cyclone.clientowner
           \
3      --resources=/opt/jboss/keycloak/target/cyclone-
           clientowner.jar \
4      --dependencies=org.keycloak.keycloak-core,
5      org.keycloak.keycloak-services,
6      org.keycloak.keycloak-model-jpa,
7      org.keycloak.keycloak-server-spi,
8      javax.ws.rs.api,
9      javax.persistence.api,
10     org.hibernate,org.javassist"
```

Listing 5: Command to add a new custom extension into Keycloak's Wildfly server. It requires to list the name of the package and all the other package dependencies.

Then, we can add our module into the list of providers in the Keycloak's configuration and the platform will load our extension the next time we start the server. However, executing this, may not be feasible, as we want to provide an easier way to deploy rather than having to download and install Keycloak and then adding the custom module. To avoid so, we have created a custom Docker image of Keycloak,

[4] http://blog.keycloak.org/2016/12/considering-removing-mongo-from-keycloak.html

which includes inside our custom extension. We have created it extending Keycloak version 2.2.0 Docker image, just to include our module. To do so, we can create a simple Dockerfile that will automatize the image craetion This, allows a user to use a simple *docker-compose* to start a new CYCLONE service which includes all the different components. As our project is an open source one, we can upload it to Docker Hub for free.

However, we are aware that creating the Docker image is not a fast task and it could be automated. Thus, we have created an automatic build and deployment to Docker Hub using Travis CI. This continuous integration setup fetches the code when committed to the master branch of project and then it installs and packages the extension in a JAR file. Then, using the project's Dockerfile we create the image and we upload it to Docker Hub. This, allows us to recreate a new image in terms of few minutes, just commiting new data to the git repository. For example, we could create a new updated version of our image with recently released Keycloak version just updating the version number in the Dockerfile.

All the source code needed to compile the extension can be found in this extension's GitHub repository [5]. The implementation described in this thesis can be found in commit *354ae0c*.

[5] https://github.com/cyclone-project/cyclone-client-registration

5 SSH login integration

As described previously in UC1 and UC2, scientists require to access their secure servers via SSH. However, setting up an SSH RSA key is not an easy task and it is time consuming. In this subsection we introduce a solution that we have called CYCLONE-PAM where users can log in into a SSH server while using their SSO credentials from their own institution. The implementation is based on extending the PAM workflow to allowing to use OIDC as authentication source and the main objective is to provide a simpler system to login to a terminal. In this subsection we provide an overview of the motivation to implement this solution together on the extendability of SSH. Then, we describe what is the objective architecture of our system. Next, we explain the implementation of the chosen solution using Linux PAM modules configuration while using CYCLONE's OIDC as authentication backend. Finally, we provide some insights on future work, and how is the current work on a graphical client for the solution is evolving.

5.1 Motivation

As described in the use cases UC1 and UC2 of CYCLONE project, one of the user's needs is to be able to access the deployed cloud. This cloud contains the set of tools, platform or services that the users need to work with. As per the requirements, this services need to be secured, in this case by the SSO provided by CYCLONE platform.

One of the possible ways to connect to the server is via SSH. SSH is used in the provided use cases to setup remote shell connections, virtual SSH tunnels and initiate X11 remote desktop connections with a server. SSH allows to authentication in three different manners [28]:

1. Username and Password: a simple username and password connection that is validated against the internal set of users of the server.

2. Username and SSH RSA Key: a username plus an RSA public key that allows to encrypt and authenticate a user without the need of a password.

3. Keyboard-Interactive: a generic authentication method that can be extended to implement custom authentication mechanisms. Any authentication system that requires the user's input can be performed using this method. This include: password, PAM, RSA SecureID and RADIUS.

Even if using an RSA key login has lots of advantages such as high level security and not having to provide the password every time, the downside is that creating the key and deploying it in the server dynamically is not so easy, at least for people without IT background. For the use case of scientists, this is not so feasible, as you need rather advanced IT skills in order to setup the SSH RSA client in your computer and create the key, plus command line needs to be used. With CYCLONE we want to simplify the login process as much as possible, so users can focus in their work rather than login in into the system.

The solution, CYCLONE-PAM, we have devised for this problem is to *allow the users to login into the remote server using their own eduGAIN account* through the use of CYCLONE platform. Through the use of Keyboard-Interactive authentication method, we can create a custom authentication which can be connected to the authentication system provided by CYCLONE.

5.2 Architecture

In order to allow users to login with their eduGAIN account, we need to somehow connect and provide authentication through CYCLONE and then forward the validated user's data to the secured server. OpenID Connect provides multiple authentication flows which may or may use a browser to authenticate the user. In our case, and with the purpose of simplifying the authentication system for the users, we need to do authentication via browser.

Also, browser authentication is a requirement because of SAML 2.0: the SAML 2.0 IdP from eduGAIN that Keycloak uses can only work using a browser workflow, as described in the SAML 2.0 protocol definition. As a result, this means that Keycloak needs to do an OIDC callback to the client with the user's information and that requires the secured server to have an HTTP server listening.

A general overview of the authentication process can be seen in Figure 11:

- The users logs in into the server via SSH (1), which triggers the PAM authentication (2) and starts an HTTP server using python in a random port (3).
- The URL to the HTTP server is forwarded back to the user (4 and 5).
- The User connects to the provided URL, received through the console, (6) and gets redirected to the authentication login in CYCLONE (7).
- Authentication happens online in the browser. Keycloak saves cookies in the user's browser for future use and faster login. OIDC POSTs a callback to the HTTP server with the validated user data (8).
- The HTTP server then forwards the data to some authorization logic which will define if the user is authorized to access the secure server or no. The result is returned to SSH that in turn will allow or not the access to the user.

As seen, we need to extend the SSH server with custom logic to define if the user is authorized or not using the user's authenticated data, plus we need to spawn an HTTP server that will redirect to the specific authentication endpoint of the OIDC protocol and also receive the POST callback from OIDC. Redirection is the choice to access to the authentication website because shortens the URL that the users have to input in their browsers.

We inspired this design on previous work done in crowd_pam module [29], which allows to login people using Atlassian Crowd as backend. However, we differed in multiple implementations as for example we do not allow to create new clients if they do not exist.

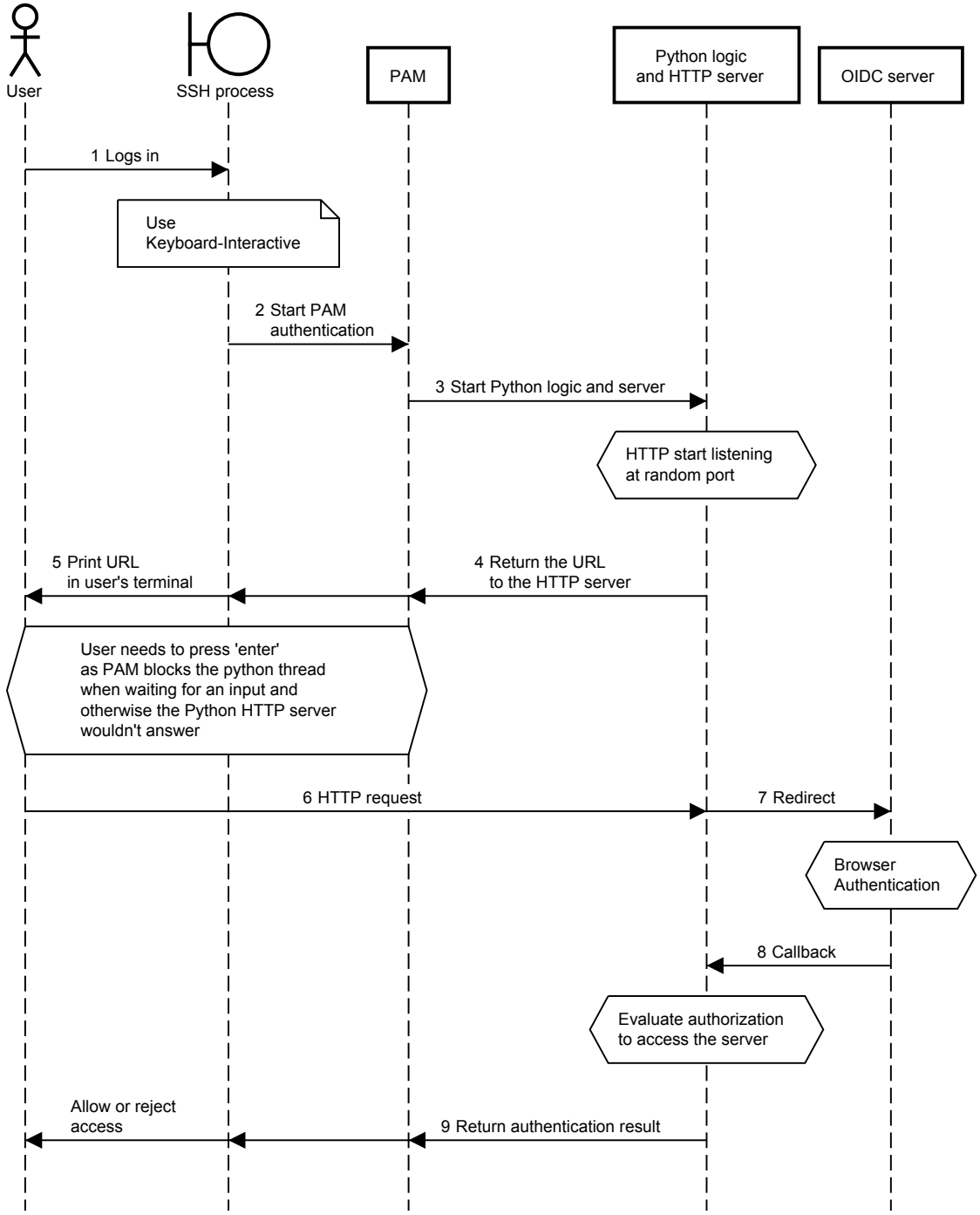

Figure 11: UML sequence diagram representing a general overview of the whole SSH login integration with OIDC

5.3 Implementation

Following the motivation found in the previous subsection, we need to find an extension to SSH so authorization can be done through the use of an external OIDC backend. After reviewing the different authentication systems provided by SSH [30][31], it is clear that the most flexible one is using the Keyboard-Interactive login and using the PAM subsystem for authentication purposes as it allows custom configuration in the login workflows. However, still there is the challenge to authenticate using an OIDC server from a PAM module.

The implementation is separated in three main parts, implemented in Python:

1. PAM module interface (subsection 5.3.1).
2. Authentication via OIDC (subsection 5.3.2).
3. Authorization procedure (subsection 5.3.3).

To do so we researched what are and how we can implement PAM modules, as described in subsection 5.3.1.

5.3.1 PAM Module Implementation

Linux Pluggable Authentication Modules (PAM) [32] provide a plugin framework which grants custom authentication support to any service or application found in a Linux or GNU/kFreeBSD environment. The PAM standard is known as DCE-RFC 86.0 and a copy of this standard along with the System Administrators' Guide and Module Writers' Manual and Application Developers' Manual is found at `/usr/share/doc/pam-version#` inside most of the Linux based operating systems. By using this plugin system we can extend the existing authentication subsystem with custom features.

By default SSH uses the internal Linux PAM subsystem of user accounts to authenticate the users trying to use the Username and Password method. By using Keyboard-Interactive we can choose to still use the PAM system to authenticate users while setting up a custom form to fetch the data from the users. Then, we can extend PAM to allow authentication through OIDC, the simplest and clearest way to implement custom, more complex code is to use *pam_exec.so* library. *pam_exec* allows to execute custom Bash scripts and implement more advanced logic inside PAM authentication. As PAM depends on boolean results, the custom shell script only needs to return true or false values. However, there is no existing implementation of OpenID Connect client in Bash, and creating it for this project would be a huge overhead.

As implementation of PAM modules is usually done in C and creating a simple HTTP server in this language is not so easy, we researched other approaches. Other options in terms of languages were Ruby and Python. Both languages allow to setup an HTTP server with a minimal amount of code. Also, Python language could be used to define the advanced authorization logic of the PAM module after successesfuly authenticating the user via OIDC. Avoid implementing the authentication logic

in C together with another language for the HTTPS server would also reduce the complexity of CYCLONE-PAM.

We analyzed other different solutions to implement a PAM module, using other languages. We found two possible options:

- pam-python [33]: written in Python, allows to implement custom PAM modules. Last commit dates from 2016. Also, has some examples and documentation, plus checking the issues, the main developer still is maintaining the code and possible issues.

- ruby-pam: written in Ruby, also allows the same features as pam-python. However, it is not actively maintained and there is barely activity in the repository since 2012, from when the latest update dates of.

Clearly in terms of maintenance, the Python module is more actively developed, which gave a clear decision on which of the two options to choose: python-pam module.

Pam-python module exposes the PAM implementation interface to the Python interpreter. According to the author of pam-python, "writing PAM modules from Python incurs a large performance penalty and requires Python to be installed, so it is not the best option for writing modules that will be used widely". However, our use cases determine that we do not have many users using the PAM login in parallel too often, which means that performance is not a critical point of our implementation.

On the other hand, Python secures our implementation against problems caused by memory allocation or corruption and is also generally shorter to write it in C than in Python. As the memory corruption would happen in PAM, any corruption could affect any program using the PAM subsystem. This makes pam-python an ideal solution for basic implementations of the PAM API while minimising the danger of introducing memory corruption into any program using PAM.

The only drawback found in this module is that as it depends on the PAM protocol, there are some limitations caused by PAM when executing the script:

- *The python scripts freeze when waiting input from the user.* For security reasons and internal workings of PAM, the executed code needs to stop when user input is requested. This means that python will stop being executed, even if ran in another thread.

- *Notifications and messages require user input in order to be forwarded to the user's console.* If PAM forwards messages to the user, if they are not followed by a user input (such as press a button) afterwards, the message won't be forwarded to the user.

This means that it is a must for for the user to press a button when information is shown back to the user, otherwise any python script won't be executed. This can be considered as loss in user experience as we require more actions from the user so the implementation works. However, this also increases the security as we make sure that no script has started the login in behalf of the users, without their knowledge.

pam-python is not just a Python module but also depends on a library that needs to be installed on the OS and that works as a bridge between Python and the C implementation of PAM. It can be installed with `apt-get` or can be compiled from the source code found in its repository. At the moment, there are only prebuilt packages for Debian based distros such as Ubuntu or Linux Mint. The default location of the library after installing it with the DEB package is `/lib/security/pam-python.so`. However, as most of the images used in the genomics case are based on CentOS they are not compatible. We tested doing a build in CentOS and it can be compiled and executed without any problems in this other family of Linux distros.

Another possible limitation that might be problematic on using *pam-python* is the operating system's Python version. Pam-python is compatible with Python versions 2 and 3, though other packages needed to implement the OIDC authentication may require a minimal Python version higher than the one shipped with the OS that the server runs. However, Python can be updated via portback packages or compiling it from source code and replacing the system's Python command, as PAM uses the Python executable loaded by the system.

Now, the next question that arises is how do we load this PAM module into the PAM configuration of the system. Each PAM enabled service has its own PAM configuration found in the `/etc/pam.d/` folder [34]. According to its description each of these files "contains a group of directives that define the module and any controls or arguments with it." In our case we need to extend the authentication module interface of PAM to update the authentication of the SSH PAM module.

The original workflow for SSH login consists on first checking if there is an RSA key available. In case it exists, it will authenticate the user using it. Otherwise, it will try an authentication via username and password using PAM. In the end, we are only replacing the username and password authentication with our own custom PAM module, the RSA key login is still available beforehand. SSH's default configuration uses the common authorization configuration (username and password) to authenticate the user:

```
1 # /etc/pam.d/sshd
2 # PAM configuration for the Secure Shell service
3
4 # Standard Un*x authentication.
5 @include common-auth
```

Listing 6: Default configuration provided in SSH's PAM module. It loads PAM's common authorization (username and password) and uses it to authenticate the user.

In our implementation we replaced the common authorization flow with our custom PAM module:

```
1 # /etc/pam.d/sshd
2 # PAM configuration for the Secure Shell service
3
4 # Standard Un*x authentication.
5 # @include common-auth
```

```
6 # module_interface      control_flag      module_name
      module_arguments
7   auth                  required          pam_python.so
      cyclone_pam.py
8   session               optional          pam_python.so
      cyclone_pam.py
```

Listing 7: Updated configuration in SSH's PAM module. Now, we are loading our custom PAM module and settings through the pam_python.so library. Also, we set it as required as it is a must that this validation goes through in order to approve the user as authenticated.

As we know, different users will have different needs, and thus different requirements in configuration. Global configuration of the module can be loaded via submitting an extra argument to the module_arguments containing the path to the configuration file:

```
1 auth required pam_python.so cyclone_pam.py /lib/security/
    cyclone_config
```

Listing 8: SSH'S custom PAM configuration including custom configuration loaded into the PAM module.

In our case, we have set up the default configuration location in `/lib/security/cyclone_config`, though its path can be customized. The reason is that it is the same location where the pam_python.so library is located and shortens the length of the module arguments. By default, if a path to the configuration is not provided, Python loads the default configuration. At the moment, the configuration only contains the set of ports that PAM module can use to expose the HTTP server in, which are used later when setting up the HTTP server in subsection 5.3.2.

With this configuration loaded, PAM executes a Python script to setup an HTTP server to authenticate the user and fetch data from him.

5.3.2 Authenticating Against OIDC and Fetching User's Data

A HTTP server is exposed by using Python's SimpleHTTPServer module, executed directly from the Python script that evaluates the PAM authentication. The port to be exposed to is defined by the configuration provided to the PAM module. The configuration contains an array of ports and/or arrays of ports indicating where the HTTP server should the service use. As there are server clouds and other locations where the available ports may be limited, these settings allow to define which are the possible exposed ports in the firewall so the HTTP authentication login can work while connecting from the user's computer.

Also, as multiple users can try to login at the same time, a random port from the defined set of the available ones to connect is chosen. This allows multiple users to connect at the same time plus increases the security, as an attacker wouldn't know which would be the used port to see the login webpage.

The SSO authentication happens in the OIDC server. The first time a users log into the SSO server they will be prompted to login using their credentials. If successful, the SSO website will save a token in the user's cookies, so next time he logs in, credentials do not need to be entered again. As a result, pasting the given URL into a browser containing the SSO cookie will validate you automatically and will log you in into the server. This decreases the amount of interactiosn of the user with the server and speeds up the login time.

After the authentication is successful in the OIDC server, a callback is received by the secure server's HTTP endpoint. The endpoint of the callback is provided when redirecting the user from the secure server HTML to the OIDC authentication service. The callback has to be done directly to the secure server, so we need to provide either an IP or a domain name. An IP is not viable as we cannot be sure if the IP is public or internal. Otherwise, to use a domain name we need to know which one to use. By default CYCLONE-PAM uses the hostname of the server. In the future, this should be a configurable setting of the module configuration.

Within the success callback a JSON Web Token (JWT) with the user's basic validated information is included. Then, following the recommendations provided by the OIDC protocol, we can validate the authenticity of the JWT with the server's key and extract the data from it. We use the python-JOSE python package to handle the JWT. This package depends on the PyCrypto implementation to decrypt the JWT signing. Also python-jose requires a minimum Python version of 2.6 to work, which defines one of the minimum requirements of our solution. To validate the data we need to use the key provided by the OIDC client. At the moment this is achieved deploying manually the key to the server. In the future, it should be considered to either create an easy setup to deploy the validation key or to use instead the client id and client secret to authenticate against the OIDC server.

From the extracted user's attributes we need the email. In case the email is not found in the token, we try to fetch it from the OIDC user's endpoint. eduGAIN does not enforce a set of attributes for the users as it only recommends to have them, so some IdPs may not expose them and eve if they do so it is quite possible that the attributes may use different key values. For this reason, we try with different aliases of the mail attribute such as 'email' or 'mail'.

After authenticating the user and (if needed) fetching its information, we need to validate that this SSO user can login as the requested user in the secure server.

5.3.3 Local authentication and authorization logic

The local authentication and authorization process allows the CYCLONE-PAM to determine which remote users can login as specific users in the secure server. When logging into the server via SSH, users need to indicate with which local username of he wants to log in. However, we may not want users to login as any possible user in the system, as they could login as root user and this would expose deep security risks. It is not feasible to create one user per SSO user, as use cases require sometimes that users share content and access to the data, plus allowing the script to create new users exposes even more security risks. Having more users in the server would only

increase the complexity in terms on how to update the permissions of many users at the same time.

What is needed to avoid this problem is to create a system which can allow us to map the users login in via SSO to the local users of the secure server. This approach needs to be simple and flexible, so it adapts to the needs of access of each deployment.

To do the mapping we need to rely on a constant attribute of the user that can allow us to track who is that user. In CYCLONE, we cannot store the data of the user in the Keycloak server because of stackeholders requirements and we need to delete all the user's data from the Keycloak database. That stops us from creating a unique local ID in Keycloak that could be used to track the user and forces us to use the user's attributes to track who is that user. Also eduGAIN, as explained before, does not enforce SAML attributes so we need to rely on the most used ones. This is why we chose the 'email' attribute, as it is unique, not so critical to share and allows us to identify each of the users individually.

Having chosen which attribute we want to use to identify our user, now we need to define where to store this value securely. Our storage decision consists in adding a simple .edugain file in each of home folder of the local users of the server – /home/username/ –. In case of the root user, it is found in the root user's home folder in /root. This file is a JSON formatted file which contains a list of mails that are allowed to login in as the folder's owner.

The structure of the .edugain file is as follows:

```
{
  "users": [
    "mail1@example.com",
    "mail2@example.com"
  ]
}
```

Listing 9: Structure of a .edugain file, containing a JSON which defines the different mails of the users that can login.

The authentication logic steps in Python, also depicted in Figure 12 are then:

1. John Doe wants to login as user 'randomuser' so he does ssh randomuser@host.
2. When doing SSH he logs into the provided URL with his EduGain account.
3. CYCLONE-PAM fetches John Doe's mail from his EduGain account data.
4. CYCLONE-PAM opens the file /home/randomuser/.edugain and checks that the mail provided by EduGain matches with the one found in the file. This data is used to evaluate a match between the user's mail found in the .edugain configuration and his validated mail.

If at any moment the file does not exist or the mail is not in the file, the authentication is rejected. A mail can be contained in configuration files of different users, meaning that with your SSO account you can login as multiple users of the

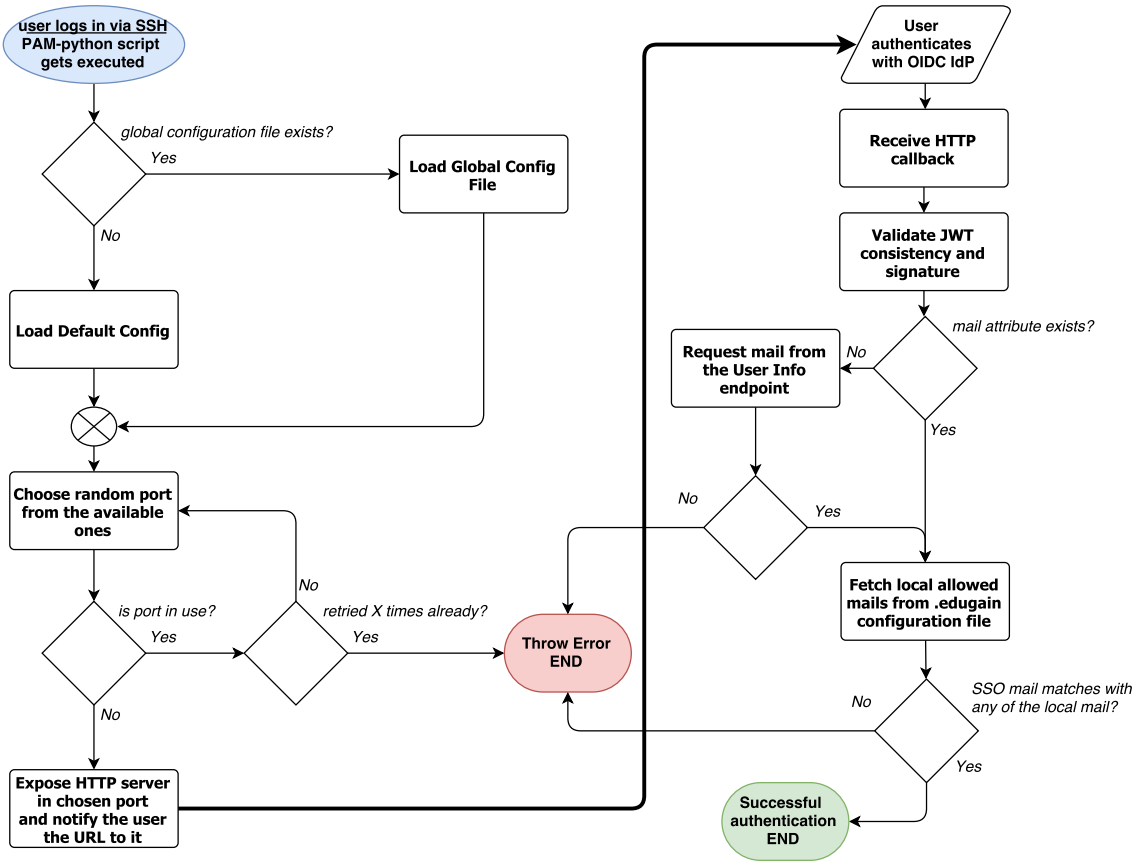

Figure 12: UML activity diagram representing the decision steps in the authorization process within the Python script

secure server. Configuration files are structured in a way that they can be extended with future settings and attributes in an easy and flexible way.

5.4 Deployment

In the previous subsections, we have detailed the decisions and inner workings of CYCLONE-PAM. However, we need an easy deployment which includes all the configuration needed for it to work. As a prototype and in order to do a fast deploy, we have created a Bash shell script that deploys the needed software by itself. The script install all the required dependencies of the system and also the modules required by the python script. Also, it updates the configuration and deploys the scripts, key and configuration to /lib/security. Finally, it does a cleanup and reconfigures SSH PAM configuration to use our custom authentication system. As sysadmins may would like to revert back the changes, it creates a backup of the replaced configuration in a new file in the same folder.

The script has been tested to work properly under Ubuntu 14.04. Newer versions of Ubuntu are still not supported but some basic changes in the script should allow using the software in them. In Fedora based systems such as CentOS, the location

of the files to be updated need to be changed in order to make it compatible with the OS.

This script can be executed from a one liner bash command that downloads it and executes the whole installation for easier deployment. Even though the best solution would be creating a DEB or RPM package which would automatically install the files in their proper locations, this simple script allows us to deploy easily this proof of concept.

5.5 Results of the Implementation

After the analysis and implementation of this solution, the result is an easy to setup and use authentication system that can allow users to login using SSO technologies. Even if the logic can make look this system slow and tedious for the user, it is in fact a fast and secure system. The procedure to login for the user is the following one:

1. User starts SSH connection to the secure server and secure server answers with an URL into the users terminal (Figure 13).
2. User opens the URL and logs in with his eduGAIN SSO account (Figure 14). Also needs to approve the consent that allows us to fetch the user's mail (Figure 15).
3. User gets notified about the successful login and gets access to the server's shell (Figure 16).

```
[sturgelose@Eriks-MacBook-Pro:~$ ssh sturgelose@gilgamesh
Starting the server
Browse to http://gilgamesh.snet.t-labs.tu-berlin.de:34570 to login
<Press enter to continue>
```

Figure 13: Screenshot showing the notification shown to the user after starting SSH login

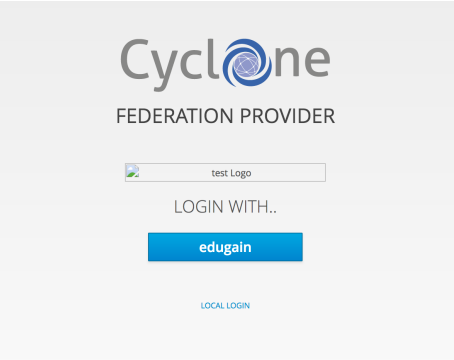

Figure 14: Screenshot showing the CYCLONE login page after the redirect

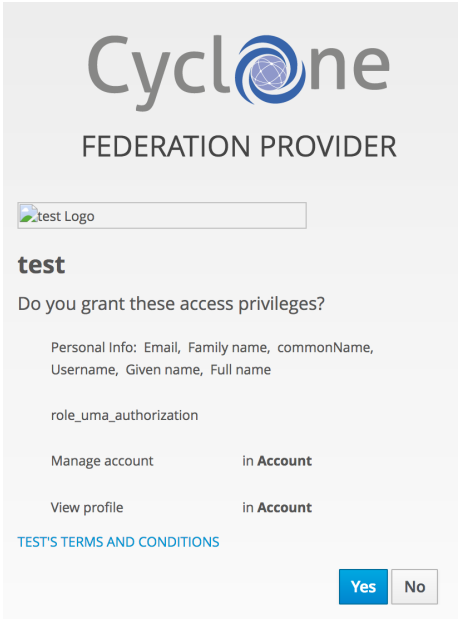

Figure 15: Screenshot showing the consent approval in OIDC

Authentication Successful!

Please, go back to the terminal to finish the login
This windows will automatically close in 5 secondes.

Figure 16: Screenshot showing the successful authorization notification in the browser

```
[sturgelose@Eriks-MacBook-Pro:~$ ssh sturgelose@gilgamesh
Starting the server
Browse to http://gilgamesh.snet.t-labs.tu-berlin.de:34570 to login
<Press enter to continue>
User has been authenticated in eduGAIN network
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-31-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
```

Figure 17: Screenshot showing the approved login terminal in OIDC

When logging in through the browser, the SSO token is stored as a session cookie. OIDC can load automatically the cookie if found in the browser session, and automatically authenticate the user towards the secure server without the need of him entering again the credentials. This allows to decrease even more the login time and improve the user experience. In this case, the behaviour of the OpenID Connect token would comparable with the one of a RSA key when logging in via SSH.

However, the main difference is that the OIDC cookie has expiry date, and will need to be recreated after the browser is closed or the session expires. Also, the token is not stored in the cookie and is given directly by the SSO server to the client in the secure server. As this token is generated and signed dynamically every time the user logs in into the browser and has expiration date, we could say that this increases the security of the login. Another benefit is that we are not storing the token in the user's computer and it is instead hosted in a server that is trusted by other services and secure enough to host the data of many users.

Another advantage is that our solution does not require the user trying to access the server and the one authenticating being the same person. As the SSO authentication happens in parallel, a user (U1) would be able to login in behalf of another one with access to the secure server (U2), just sharing the URL to the HTTP server provided in the terminal. U1 would be able to login in behalf of U2, without knowing any credentials. Also, U1 wouldn't be able to login again without another authentication of U2.

Yet this creates a new problem because of Cross Scripting issues: if a malicious user achieves an authorized user to open the login URL, the authentication might get automatically approved via the SSO and the unwanted user would get access to the server without the user even knowing so. Still, this problems are usually covered by CORS security implementations in the browser and in the server, that avoids unknown links being loaded without the user's consent.

Another security concern is that in secure data centers the ports available are limited, usually by strict security rules in the firewall. This makes the set of ports where the HTTP server can be exposed quite limited. As a result, with a reduced number of ports a malicious user would be able to easily find in which port the HTTP server is running and attack it. Of course, this doesn't really expose directly a security problem, but can be an easy entry point for future attacks.

Finally, another problem is the compatibility with actual SSH agents. Because

we are using Keyboard-Interactive input for SSH, usual SSH clients may not work as expected. Even though SSH will still accept SSH RSA keys, login via username and password is not possible after integrating CYCLONE-PAM into a server. This, for example will affect and break X11 clients that rely on the SSH protocol to sign in into the remote server.

The code of the implementation described in this subsection can be found in its own public GitHub repository [6]. All the implementation described for this component are committed as per commit *707266e*.

5.6 Future Work

Future work of this implementation generally tries to cover the issues described in the previous subsection in an attempt to improve even more the security of the module but also is required to improve the usability of it.

First, CYCLONE-PAM only supports JWT client authentication. One simple way to extend it is with the use of client_id and client_secret which can be used to authenticate the client. This would also simplify the deployment as this will eliminate the need to deploy a key to validate the JWT.

Another needed improvement would be using HTTPS in the SSO callback. At the moment the callback is done via plain HTTP, and it includes private data of the user. To do so, a HTTPS certificate should be deployed in each of the servers, which, increases the maintenance difficulty. It could be at least a self signed certificate, that is added to the trusted certificates store of the SSO server. Another possible solution would be using a reverse proxy that would de-encapsulate HTTP to HTTPS.

One of the problems in the previous subsection, describes how a user could trigger an authentication in the secure server without knowing thanks to CORS vulnerabilities. This situation could be easily prevented with a timeout in the login. Setting that users have to login in under 60 seconds, gives no time to a malicious user to force someone else to use the link and reduces the threat.

Finally distribution of the software should be improved. For starters, CYCLONE-PAM should be packaged in a DEB or RPM bundle to improve its upgradability. Also, the locations of configuration files should be rethought so they are found in the proper locations of the filesystem. Also, one possibility to improve the user experience would be to add an extra step during user creation in the server to allow to introduce then which are the allowed emails that can login into the account using CYCLONE-PAM.

As a summary, there are many possible improvements in the software to increase its reliability, but providing that the actual implementation is a proof of concept, it is easy to find so many ways to extend its functions and security.

5.6.1 Electron Based Desktop Client

As Keyboard-Interactive SSH logins are not so common, we have found out that usage of desktop clients which rely on the default SSH protocol configuration using

[6] https://github.com/cyclone-project/cyclone-python-pam

username and password pair may not work after applying our solution. Specifically, this is troublesome when dealing with X11 clients.

To avoid this kind of problems and simplify the usage of CYCLONE-PAM, we are currently developing a desktop client based on GitHub's Electron platform which allows to create SSH console connections, SSH tunnels and also X11 remote desktop connections using XPra. This client is currently in development and at the moment it can already create all the different SSH connections needed and X11 remote desktop connections secured via SSH tunnel, only in Microsoft Windows, though.

One of the advantages of using this client rather than directly copying the given URL in the console, is that we can do the login directly in the client. Because the electron framework is based on Chromium web browser, we can directly open the login URl for the user in the same window. Also, we can keep the cookies in the browser window as long as the client desktop is running, so having it running as one of the operating system's services, we can allow the user to authenticate in different secure servers just signing in into the SSO login page just once.

6 Evaluation

In this chapter we evaluate the results of the implementations described in Chapters 4 and 5. We first evaluate each of the solutions implementations individually and finally analyze the result of the integration between both components.

6.1 Dynamic Client Registration

To validate the implementation we have achieved as described in Chapter 4, first, we individually validate that each of the requirements of the use cases is covered through the use of our extension. Next, as it has not been tested by our use cases, we compare our implementation to the updated version of the OpenID Connect client registration API that Keycloak 2.4.0 includes.

6.1.1 Use Cases requirements Validation

In subsection 2.3, we listed a summary of all the different requirements that we would need to fulfill when creating our solution. In this subsection we analyze that we cover each of the requirements previously described:

- *R1*: During all the design of these solutions we have focused on security. To do so, we have based our implementations in existing frameworks and relied on industry standard SSO technologies. The only thing that should be analyzed is to secure the connection of the PAM module properly using SSL certificates.
- *R2*: We have covered the need of securing different interfaces through the use of the PAM module. With it we can secure not only the web interfaces, which can be simply protected using OIDC or SAML, but also SSH connections and with it tunneled connections through SSH such as X11 remote connections.
- *R3*: In the PAM module configuration we allow settings to make sure that isolated networks with strict firewalls can be compatible with our system.
- *R4*: To allow our system to be compatible across multiple clouds, we have implemented our software using Docker as a a base. This allows to run CYCLONE platform in any cloud independently of the operating system. CYCLONE-PAM, we have the source code and we can compile the required libraries in any unix SO, which even if it is some extra work, it would allow to make it run in any platform. Login into the platform can be done as long as the user has a SSH client.
- *R4*: This has not been totally implemented, as we still require the logic to integrate our deployments with Slipstream, the virtual machine and services orchestrator that CYCLONE uses. The results are discussed in subsection 6.3.
- R5: This regulation is easily done through the use of mails to validate the authorization in the PAM module. Also, as CYCLONE provides us access to

eduGAIN's federation login credentials, we can allow access to anyone using this federation in our sytems. Further federations could be added including them in the SimpleSAMLphp proxy or either adding them into Keycloak as Identity Provider.

6.1.2 Comparison with Keycloak's Version 2.3.0 Registration API

In our background analysis of the registration API, we analyzed the different features that Keycloak's version 2.0.0 (released on 30th June 2016). At that moment in time, the registration API was already functional but in early stages in terms of features. It basically provided a system to create clients, but without the availability to set restrictions on the new clients or any sort of configuration aside of the amount of times that the initial access token can be used.

Since version 2.3.0 [22], Keycloak's client registration API provides a more granular configuration set than in previous versions. This new configuration settings allow to define the policies that limit who can create a new client and which settings a user can set when creating a new client. It also introduces two different workflows to authenticate a user before being allowed to register a new client in comparison to the one single anonymous access in previous versions. Each workflow can have its own set of policies. This two workflows are:

- Anonymous Access

  Anyone, without the need to authenticate using a Bearer Token or Initial Access token can create a client. The trust relationship is nearly non-existent in this workflow, thus, this client creation method is quite restricted and limits the usage to a set of trusted hosts and incorporates many other policies.

- Authenticated Access

  The user creating a new client needs to be authenticated using either an Initial Access Token or a Bearer Token. An Initial Access token is an anonymous token that allows to create clients and behaves in the same way the Initial Access Token in version 2.0.0 worked. A Bearer Token, is a token that can be obtained from the token endpoint of the SSO, and is issued in behalf of a user or Service Account. In order to be able to create a client using a Bearer Token, the user needs to have a "create-client" role specific to the realm that wants the client to be in. However, this only allows to create clients, again limited by by the set of policies for this workflow. This same role allows users to create clients through Keycloak's administration API. However, to edit clients, a user requires a "manage-clients" role. This role allows complete access to change settings in any client of the realm. As discussed before in subsection 4.1, granting the "manage-clients" privileges to someone would allow to a user to escalate and get superadmin rights. Another similar role is the "view-clients" which would allow you to see any of the attributas and settings of any client. this includes the secrets, JWT and any information that would allow you to use the client in behalf of someone else. Also, not allowing to edit client permissions

would decrease the self-service and require the administrator's help to change basic changes. In general this roles, even if provide the needed access, they can provide a too wide access to the system and thus drastically increase security concerns. Otherwise, we are only limited to create a client, and see the result of our creation, without ability to change settings.

Both of this workflows include a set of configurable policies that allow or disallow a user to create users, including: list of Trusted hosts, if a Consent approval is required, the allowed scope of users that the client can authenticate, the maximum of clients that can be created, the allowed SSO protocols that can be set and which client templates can new registered clients use. Thus, this new version provides an extra collection of settings that improve the user experience and flexibility of the registration API.

Still, we need to compare the advantages and disadvantages of using each of the solutions to evaluate our system against Keycloak's new implementation. To do so, we have compared both APIs in different aspects:

- **Authentication workflow(s):** comparison of the different methods that allow us to authenticate against the API

  - Keycloak API: there are multiple workflows. Either we can authenticate with a Bearer Token, with an Initial Access Token or directly without authentication. This allows that mostly anyone could create a client without many problems in an easy and flexible way. However many authentication workflows also means multiple kinds of tokens that can be issued and more complexity.
  - CYCLONE-API: there is only a single workflow, which is equivalent to the Bearer Token Workflow in Keycloak's registration API implementation. This provides less flexibility but at the same time increases the simplicity, as it is exactly the same flow used to login into Keycloak's administration API.

  Both solutions include a Bearer Token workflow. However, Keycloak's API allows more workflows and other kind of users to be able to login.

- **Security during authentication:** comparison on how we make sure that authentication credentials are valid that its implementation is flawless.

  - Keycloak API: provides an implementation developed by Red Hat and has multiple tests and an open source community validating the security of the platform. Relies on Jboss' Firefly platform under the hood to secure all the transactions.
  - CYCLONE-API: uses exactly the same authentication code as Keycloak's administration API, which results in an implementation as secure as Keycloak's. Tests are not implemented against this endpoint but can be implemented using Keycloak's as inspiration.

- **User tracking:** mechanisms that allow to track who is creating each client or who has rights in each client.

  - Keycloak API: as the workflows can be anonymous, there is no real way to check who is registering a client or who has a token that allow to manage a client. One only possibility would be matching API calls with tokens requested somehow from the logs that Keycloak exposes.
  - CYCLONE-API: because the only authentication workflow requires an authenticated account, all the users need credentials in the SSO platform. Thus, we can track who is who when requesting new clients. The client registration events could be exposed into the UI interface in the log subsection thanks to the events that get executed whenever someone interacts with the API endpoints. Also, our implementation includes an owner-client relationship, where all the clients created through this API can be mapped to someone responsible for it.

  Keycloak's implementation does not provide any form to track who is who, while our implementation focus is to exactly solve this problem. Thanks to this, we can distribute responsibilities throughout a set of users without handling them access to all the possible client resources.

- **User roles required:** which roles and privileges a user needs to have in order to handle CRUD operations with the clients.

  - CYCLONE-API: the role can be customized. Having this chosen role allows users to create, view and edit clients. However, they can only view an edit the clients under their ownership. Also, it is disallowed to edit the roles given to the service account.
  - Keycloak API: to create clients, user requires the realm's "create-client" role. To view and edit a user needs to have the "view-client" and "manage-clients" respectively. However, this both roles provide full access to all the roles in the realm.

  Our implementation provides more rights but limited to the set of clients that the user owns. In contrast, Keycloak's implementation requires too wide access if wanting to edit a client, and it's only useful to create clients.

- **Policy Management:** availability of settings that can allow us to limit which users and remote hosts can create clients and how many clients can be created.

  - CYCLONE-API: at the moment there is no policy management implemented. The only available setting is to change to which role is the API available to. As a user with the proper role you can change any setting of a client but the service account roles and create as many clients as needed.

  - Keycloak API: provides an interface from where to set different policies and its details. Through it we can define limits of the amounts of clients and which settings are allowed in new clients.

  Keycloak API provides a more granular privilege selection, and allows to set limits on how many clients can be created and from where and who. However, this is a requisite for them as they use anonymous authentication. In our implementation, all privileges are given as authentication allows us to make sure they are reliable users.

- **User Interface:** possibility to manage the different settings of the API through the use of a web UI.

  - Keycloak API: provides an integration of the API settings with Keycloak's administration console web UI. With it, all the different policies can be managed in a simple manner. Also, new Initial Access Tokens can be created or disabled.
  - CYCLONE-API: does not provide any kind of user interface to manage the settings. Could be integrated in the future with Keycloak's UI using a theme to exten its functions.

  Keycloak's implementation provide basic configuration management through the administration console, and tries to cover the objectives of Keycloak platform: to be easily configurable without the need of coding configuration. In our implementation there is no interface available, though it would be possible to create integration with Keycloak's UI creating a custom theme.

- **OIDC Standard:** status of compliance with the OpenID Connect Dynamic Client Registration 1.0 (OIDC DCR) specifications. This allows ready made clients to automatically register themselves in any SSO that is implements this standard.

  - Keycloak API: Keycloak fully supports the OIDC DCR standard and it has been certified by the OpenID Foundation certification process.
  - CYCLONE-API: after analyzing the requirements from OIDC DCR, we can say that it supports the metadata response and request models and the authentication workflow requirements provided in the specification. However, it does not fulfill the requirements of the client endpoint URL. One of the reasons is that Keycloak already uses the suggested path to locate the API's endpoint in. Also, testing the API requires paying a fee, which is out of the scope of this thesis.

  Keycloak fulfills the specifications and is built and certified to be approved by the standard. In our implementation, we succeed in completing most of the requirements in terms of authentication workflow but we are missing some part of the specifications when building and parsing the URL arguments. Though

it is not certified it should work with most of the cases when creating a new client.

As a summary, Keycloak's implementation objective is to fulfill all the requirements of the OIDC standard and focuses on allowing anonymous services to create clients, thus broadening the scope of services that can register a new client. Also, they provide a better integration with the UI. Furthermore it also has the advantage that is created as part of the core of Keycloak and can easily make all the changes needed in other SPIs to fit properly in the platform. Also it has a properly tested security suite.

In the other hand, our implementation focuses on being able to track the client registrants and diminishes the scope of users in favour of more privileges to the possible users. It might be less integrated into Keycloak as it is not part of the core modules of the platform and also it lacks UI integration for easy management. However it relies on the same tools that Keycloak's implementation does and thus can provide a similar security level. Also, it fits with the set of user requirements that CYCLONE requires and allows to offload support of the administrator in behalf of more rights to the client owners. As an objective it tries to distribute the responsibilities of a client to the owner rather than centralizing them into the administrator figure.

In conclusion, our implementation is really similar to the updated Keycloak API, but provides a better fit to the use cases of CYCLONE. It's quite probable, at least according to the discussions in Keycloak's developer mailing list that there will be a more fine grained permission system that will overlap with the implementations done by our extension, but as there is no actual solution providing our needed requirements, ours is the only implementation fitting this specifications.

6.2 SSH Login Integration

To validate our implementation we have relied on the feedback provided by one of the CYCLONE use cases after using a prototype. This has provided us essential data on how to improve the client based on the real experience of the solution. The review has been provided by the French Institute of Bioinformatics (IFB) and they tested the software in both Ubuntu 14.04, used to develop the software and fully supported and also in one of their custom images based on CentOS.

In the test performed in Ubuntu 14.04, they have found that installation through the automatic installation script provides an easy deployment. It still should support some parameters to define a path to the custom configuration. In terms of configuration, the first limitation has been that to generate the URL to the server, the module used the hostname. However, in their cloud there is no possibility to reach the virtual machine from the outside using it. Instead they suggested using the fully qualifiyed domain name (FQDN). In future version it should be considered to allow the user to choose what URI base to use from the module's configuration.

In terms of usability, there were some difficulties in the user experience. The first one is the need to press any button to run the server and unblock the python script.

This, as explained in subsection 5.3.1, is a limitation by the software and created by requirements of the PAM specifications.

Another UX issue includes the naming of attributes of the user's validated data, where in our implementation we look for the "email" key and in some implementations of the eduGAIN federation the email data of the user is found under the "mail" key. After this feedback we are now looking for both keys, though, this should be a requisite defined by the usage requirements of CYCLONE platform. Otherwise, there is no way to predict which attribute is used by all the possible identity providers of the eduGAIN federation.

Also, their cloud limits the usable ports to ports 80 and 443, thus limiting the amount of people that can login at the same time to only 2. Still, the configuration settings of the module allow to set this ports to the ones to be used to expose the HTTP server.

Finally, the module should be extended to support client_secret and client_id authentication aside of the JWT one. This would make the system compatible to the configured settings of their client. Still, it has a simple fix, and requires requesting an Access Token via an HTTP call using this parameters as authentication instead of the JWT.

In general, the feeling they have is that this system simplifies the login and provides control on the user credentials with minimal access to the server, and reliability on the user verification. Also, it increases the security as there is no need to transmit new credentials to new users neither to have them registered in a new system, as they can use their personal credentials and an already existing account.

6.3 Client Registration and SSH Integration

Until now we have described the implementation of two separate solutions. However, they together need to integrate with each other in order to provide a SSO platform to authenticate any service dynamic deployed. The integration would be that we can create a client via the registration API and automatically configure a server provisioned with the CYCLONE-PAM module. However, to do so, we need some external logic. This logic that would create the client and configure the PAM module would preferably be execute in the orchestrator that deploys the set of VM of the new service, in CYCLONE's case, in SlipStream. There, we can create a simple shell script that would do the needed calls and setup the configuration.

For easier configuration of the PAM module, we should improve its installation so we can provide configuration variables for example through the ENV variables or either create a CLI that would generate the needed configuration file having an input with the different setting gathered during the creation step of the virtual machine in Slipstream.

7 Conclusion

The objective of this study was to implement a set of tools that allow to dynamically register SSO clients in user federations. The study was descriptive, focusing on background analysis and use cases with the intention to take proper decisions to fulfill the requirements of CYCLONE platform. We centered our attention on extracting the core requirements of CYCLONE use cases and defining the existing problem that we solved in this study. Existing solutions were considered for this study and inspired the implementation design of the resultant one.

The development of both components was based on the provided user requirements defined in the objectives of this study. The registration API reasons the need of a tailor-made solution and provides a solution integrated with existing components of the CYCLONE platform. An analysis of the updatability and maintainability gives reliability to the the created module. The SSH PAM module provides a new method of authentication for SSH users, providing novelty in terms of token and key management and trying to provide a similar user experience given by RSA keys through the use of web browsers, cookies and Single Sign-On technologies. This includes innovation in the authentication services in servers, centralizing data needed for authentication and authorization of users in a single server and securing the workflows through the use of industry standard technologies.

The result provided is a set of two components provide a single solution which can interact and that can be integrated into existing environments, focusing in providing a painless user experience to non technical people. The registration API is comparable in usability to the actual existing alternatives, and relies on the use of existing implementations of the Keycloak platform.

There is also uncertainty in the future expansion and development of the registration API as it is quite possible that future versions of the Keycloak platform will integrate similar options in its core. According to discussions inside of the developer community of the project one of the focuses in version 3.0 is to provide a more granular role management in the different aspects of the platform.

In general the combination of this two tools has a great development potential in terms of new possible features and innovation.

7.1 Limitations and Future Research

A major limitation of this study is the lack of final user testing in the registration API. This limitations happened because of the long time invested in the implementation caused by the lack of documentation and examples to implement a Keycloak SPI. This then affected on the available time to test the implemented software. Instead we have had to compare our results with the actual implementation done in parallel by the Keycloak community. In future time there should be a proper user testing of the API that should give insights of the improvements needed on it.

One of the main requirements for future production use is to provide a better deployment interface for both components, thus increasing the integration between them. At the moment, part of the configuration is coded as constants in both

implementations, and would be needed to expose them to flexible tuning of the system administrator and to the variables of the deployment orchestrator platform.

A security analysis would be needed to proof the security status of environments created using the tools created in this study. In the case of the registration API it would be enough expending the existing tests for Keycloak to also target the newly created enpoints. For the SSH case, a proper modeling of possible exploits and security holes with professional tools should be performed to provide a reliable security validation to users. However, as this kind of tools would be used internally in a cloud, the firewall and secure connections inside of the federation should cover most of the use cases.

Actual deployment is done manually, at least the one done during development. The next step in this implementation should be providing an implementation with the cloud orchestration platform used by CYCLONE, "Nuv.La". This would provide a proper production case for the CYCLONE project with total integration within the platform.

Finally, the electron based client created for the CYLONE-PAM module, should be extended properly to work in the mainstream operating systems aside of Windows, including MacOS and Linux systems. Still, there is support for CYCLONE-PAM in the other OSs via the user of browser authentication. This electron client also, should need proper code cleaning and user testing feedback to make sure it is up to the standards use cases providers require.

References

[1] Y. Demchenko, C. Blanchet, C. Loomis, R. Branchat, M. Slawik, I. Zilci, M. Bedri, J.-F. Gibrat, O. Lodygensky, M. Zivkovic *et al.*, "Cyclone: A platform for data intensive scientific applications in heterogeneous multi-cloud/multi-provider environment," in *Cloud Engineering Workshop (IC2EW), 2016 IEEE International Conference on*. IEEE, 2016, pp. 154–159.

[2] M. Slawik, B. I. Zilci, Y. Demchenko, J. Aznar-Baranda, R. Branchat, C. Loomis, O. Lodygensky, and C. Blanchet, "CYCLONE unified deployment and management of federated, multi-cloud applications," *CoRR*, vol. abs/1607.06688, 2016. [Online]. Available: http://arxiv.org/abs/1607.06688

[3] F. Holzschuher and R. Peinl, "Approaches and challenges for a single sign-on enabled extranet using jasig CAS," in *Open Identity Summit 2013, September 9th - 11th 2013, Kloster Banz, Germany*, ser. LNI, D. Hühnlein and H. Roßnagel, Eds., vol. P-223. GI, 2013, pp. 106–117. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings223/article20.html

[4] J. Howlett, V. Nordh, and W. Singer, "Deliverable ds3. 3.1: edugain service definition and policy initial draft," *Project Deliverable, May*, 2010.

[5] eduGAIN technical site. [Online]. Available: https://technical.edugain.org/joining_checklist

[6] Cyclone newsletter - first edition. [Online]. Available: http://www.cyclone-project.eu/assets/images/newsletters/CYCLONE%20Newsletter%20first%20edition.pdf

[7] D. Gallico, M. Biancani, C. Blanchet, M. Bedri, J. Gibrat, J. Aznar-Baranda, D. Hacker, and M. Kourkouli, "CYCLONE: A Multi-cloud Federation Platform for Complex Bioinformatics and Energy Applications (short paper)," in *5th IEEE International Conference on Cloud Networking, Cloudnet 2016, Pisa, Italy, October 3-5, 2016*. IEEE, 2016, pp. 146–149. [Online]. Available: http://dx.doi.org/10.1109/CloudNet.2016.44

[8] W. Maes, T. Heyman, L. Desmet, and W. Joosen, "Browser protection against cross-site request forgery," in *Proceedings of the first ACM workshop on Secure execution of untrusted code*. ACM, 2009, pp. 3–10.

[9] A. Pashalidis and C. J. Mitchell, "A taxonomy of single sign-on systems," in *Australasian Conference on Information Security and Privacy*. Springer, 2003, pp. 249–264.

[10] F. Feldmann, "Binding credentials: Securing (SSO) authentication," Ph.D. dissertation, Ruhr University Bochum, 2016. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:294-45159

[11] D. Recordon and D. Reed, "Openid 2.0: a platform for user-centric identity management," in *Proceedings of the 2006 Workshop on Digital Identity Management, Alexandria, VA, USA, November 3, 2006*, A. Juels, M. Winslett, and A. Goto, Eds. ACM, 2006, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/1179529.1179532

[12] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," *CoRR*, vol. abs/1601.01229, 2016. [Online]. Available: http://arxiv.org/abs/1601.01229

[13] S. Egelman, "My profile is my password, verify me!: The privacy/convenience tradeoff of facebook connect," in *2013 ACM SIGCHI Conference on Human Factors in Computing Systems, CHI '13, Paris, France, April 27 - May 2, 2013*, W. E. Mackay, S. A. Brewster, and S. Bødker, Eds. ACM, 2013, pp. 2369–2378. [Online]. Available: http://doi.acm.org/10.1145/2470654.2481328

[14] E. Maler *et al.*, "Assertions and protocols for the oasis security assertion markup language (SAML)," *OASIS, September*, 2003.

[15] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC Editor, Tech. Rep. RFC6749, Oct. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6749

[16] OpenID Foundation. OpenID Connect | Welcome to OpenID Connect. [Online]. Available: https://openid.net/connect/

[17] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "Openid connect core 1.0," *The OpenID Foundation*, p. S3, 2014.

[18] N. Sakimura, J. Bradley, and M. Jones, "Openid connect dynamic client registration 1.0," 2011.

[19] M. Jones, B. Campbell, and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants," RFC Editor, Tech. Rep. RFC7523, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7523

[20] Y. Goland, B. Campbell, M. Jones, and C. Mortimore. Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants. [Online]. Available: https://tools.ietf.org/html/rfc7521

[21] B. Campbell, C. Mortimore, and M. Jones, "Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants," RFC Editor, Tech. Rep. RFC7522, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7522

[22] [KEYCLOAK-3666] Dynamic client registration policies - JBoss Issue Tracker. [Online]. Available: https://issues.jboss.org/browse/KEYCLOAK-3666

[23] Keycloak | Server Administration Guide. [Online]. Available: https://keycloak.gitbooks.io/server-adminstration-guide/content/v/2.4/

[24] SimpleSAMLphp. [Online]. Available: https://simplesamlphp.org/

[25] Keycloak | Server Developer Guide. [Online]. Available: https://keycloak.gitbooks.io/server-developer-guide/content/v/2.4/

[26] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in api design: A usability evaluation," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 302–312.

[27] Keycloak | Server Installation and Configuration Guide. [Online]. Available: https://keycloak.gitbooks.io/server-installation-and-configuration/content/v/2.4/

[28] SSH: Pluggable Authentication Module (PAM) Submethod. [Online]. Available: www.ssh.com/manuals/server-admin/44/Pluggable_Authentication_Module___PAM___Submethod.html

[29] crowd_pam bitbucket repository. [Online]. Available: https://bitbucket.org/atlassian/crowd_pam/overview

[30] F. Cusack and M. Forssen, "Generic message exchange authentication for the secure shell protocol (ssh)," Tech. Rep., 2005.

[31] SSH: User Authentication with Keyboard-Interactive. [Online]. Available: https://www.ssh.com/manuals/server-admin/44/User_Authentication_with_Keyboard-Interactive.html

[32] Linux-PAM Official Page. [Online]. Available: http://www.linux-pam.org/

[33] Rusell Stuart. pam-python - write PAM modules in Python. [Online]. Available: pam-python.sourceforge.net/

[34] RedHat | PAM Configuration Files. [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Managing_Smart_Cards/PAM_Configuration_Files.html

[35] A. Juels, M. Winslett, and A. Goto, Eds., *Proceedings of the 2006 Workshop on Digital Identity Management, Alexandria, VA, USA, November 3, 2006*. ACM, 2006.

[36] W. E. Mackay, S. A. Brewster, and S. Bødker, Eds., *2013 ACM SIGCHI Conference on Human Factors in Computing Systems, CHI '13, Paris, France, April 27 - May 2, 2013*. ACM, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2470654

[37] D. Hühnlein and H. Roßnagel, Eds., *Open Identity Summit 2013, September 9th - 11th 2013, Kloster Banz, Germany*, ser. LNI, vol. P-223. GI, 2013. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings223.html

[38] *5th IEEE International Conference on Cloud, Cloudnet 2016, Pisa, Italy, October 3-5, 2016*. IEEE, 2016. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7774976

[39] I. Raicu, O. F. Rana, and R. Buyya, Eds., *8th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2015, Limassol, Cyprus, December 7-10, 2015*. IEEE Computer Society, 2015. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7430473